Department of Information and Computer Science

Efficient Symbolic Model Checking of Concurrent Systems

Jori Dubrovin

Aalto University

DOCTORAL DISSERTATIONS

Aalto University publication series
DOCTORAL DISSERTATIONS 111/2011

Efficient Symbolic Model Checking of Concurrent Systems

Jori Dubrovin

Doctoral dissertation for the degree of Doctor of Science in Technology to be presented with due permission of the School of Science for public examination and debate in Auditorium TU1 at the Aalto University School of Science (Espoo, Finland) on the 18th of November 2011 at 12 noon.

Aalto University
School of Science
Department of Information and Computer Science

Supervisor
Prof. Ilkka Niemelä

Instructor
D.Sc. (Tech.) Tommi Junttila

Preliminary examiners
Prof. Armin Biere, Johannes Kepler University, Austria
Dr. Stephan Merz, INRIA Nancy, France

Opponent
Prof. Daniel Kroening, University of Oxford, United Kingdom

Aalto University publication series
DOCTORAL DISSERTATIONS 111/2011

ISBN 978-952-60-4349-4 (pdf)
ISBN 978-952-60-4348-7 (printed)
ISSN-L 1799-4934
ISSN 1799-4942 (pdf)
ISSN 1799-4934 (printed)

Unigrafia Oy
Helsinki 2011

Finland

The dissertation can be read at http://lib.tkk.fi/Diss/

Aalto University

Abstract

Aalto University, P.O. Box 11000, FI-00076 Aalto www.aalto.fi

Author
Jori Dubrovin

Name of the doctoral dissertation
Efficient Symbolic Model Checking of Concurrent Systems

Publisher School of Science

Unit Department of Information and Computer Science

Series Aalto University publication series DOCTORAL DISSERTATIONS 111/2011

Field of research Theoretical Computer Science

Manuscript submitted 14 June 2011 **Manuscript revised** 23 August 2011

Date of the defence 18 November 2011 **Language** English

☐ **Monograph** ☒ **Article dissertation (summary + original articles)**

Abstract

Design errors in software systems consisting of concurrent components are potentially disastrous, yet notoriously difficult to find by testing. Therefore, more rigorous analysis methods are gaining popularity. Symbolic model checking techniques are based on modeling the behavior of the system as a formula and reducing the analysis problem to symbolic manipulation of formulas by computational tools. In this work, the aim is to make symbolic model checking, in particular bounded model checking, more efficient for verifying and falsifying safety properties of highly concurrent system models with high-level data features.

The contributions of this thesis are divided to four topics. The first topic is symbolic model checking of UML state machine models. UML is a language widely used in the industry for modeling software-intensive systems. The contribution is an accurate semantics for a subset of the UML state machine language and an automatic translation to formulas, enabling symbolic UML model checking.

The second topic is bounded model checking of systems with queues. Queues are frequently used to model, for example, message buffers in distributed systems. The contribution is a variety of ways to encode the behavior of queues in formulas that exploit the features of modern SMT solver tools.

The third topic is symbolic partial order methods for accelerated model checking. By exploiting the inherent independence of the components of a concurrent system, the executions of the system are compressed by allowing several actions in different components to occur at the same time. Making the executions shorter increases the performance of bounded model checking. The contribution includes three alternative partial order semantics for compressing the executions, with analytic and experimental evaluation. The work also presents a new variant of bounded model checking that is based on a concurrent instead of sequential view of the events that constitute an execution.

The fourth topic is efficient computation of predicate abstraction. Predicate abstraction is a key technique for scalable model checking, based on replacing the system model by a simpler abstract model that omits irrelevant details. In practice, constructing the abstract model can be computationally expensive. The contribution is a combination of techniques that exploit the structure of the underlying system to partition the problem into a sequence of cheaper abstraction problems, thus reducing the total complexity.

Keywords software verification, distributed systems, bounded model checking, UML, predicate abstraction

ISBN (printed) 978-952-60-4348-7 **ISBN (pdf)** 978-952-60-4349-4

ISSN-L 1799-4934 **ISSN (printed)** 1799-4934 **ISSN (pdf)** 1799-4942

Location of publisher Espoo **Location of printing** Helsinki **Year** 2011

Pages 224 **The dissertation can be read at** http://lib.tkk.fi/Diss/

Aalto-yliopisto

Tiivistelmä

Aalto-yliopisto, PL 11000, 00076 Aalto www.aalto.fi

Tekijä
Jori Dubrovin

Väitöskirjan nimi
Rinnakkaisten järjestelmien tehokas symbolinen mallintarkastus

Julkaisija Perustieteiden korkeakoulu

Yksikkö Tietojenkäsittelytieteen laitos

Sarja Aalto University publication series DOCTORAL DISSERTATIONS 111/2011

Tutkimusala Tietojenkäsittelyteoria

Käsikirjoituksen pvm 14.06.2011 **Korjatun käsikirjoituksen pvm** 23.08.2011

Väitöspäivä 18.11.2011 **Kieli** Englanti

☐ **Monografia** ☒ **Yhdistelmäväitöskirja (yhteenveto-osa + erillisartikkelit)**

Tiivistelmä

Suunnitteluvirheillä rinnakkaisissa ohjelmistojärjestelmissä saattaa olla kohtalokkaita seurauksia. Järjestelmän testauskaan ei aina riitä virheiden havaitsemiseen, ja siksi on alettu vaatia järjestelmällisempien analyysimenetelmien käyttöä. Symbolinen mallintarkastus perustuu ajatukseen kuvata järjestelmän toiminta logiikan kaavoilla, jolloin toimintaa voidaan analysoida käsittelemällä näitä kaavoja laskennallisen logiikan työkaluilla. Tässä työssä päämääränä on parantaa symbolisten mallintarkastusmenetelmien ja erityisesti rajoitetun mallintarkastuksen tehokkuutta, kun tarkastellaan laajasti rinnakkaisten ja dataa käsittelevien järjestelmämallien turvallisuusominaisuuksia.

Väitöskirjassa saavutetut uudet tulokset voidaan jakaa neljään aihealueeseen. Ensimmäinen liittyy UML-tilakonemalleihin. UML-kieltä käytetään laajasti teollisuudessa ohjelmistopohjaisten järjestelmien suunnitteluun. Tässä työssä määritellään tarkka suoritussemantiikka luokalle UML-tilakonemalleja sekä mallien automaattinen käännös kaavoiksi, mikä mahdollistaa UML-tilakoneiden symbolisen mallintarkastuksen.

Toinen aihealue on jonoja sisältävien järjestelmien rajoitettu mallintarkastus. Jonoja käytetään usein esimerkiksi viestinvälityksen mallintamiseen. Työssä laaditaan erilaisia tapoja kuvata jonojen toiminta kaavoina, jotka hyödyntävät modernien SMT-ratkaisimien ominaisuuksia.

Kolmas aihealue on mallintarkastusprosessin nopeuttaminen symbolisilla osittaisjärjestysmenetelmillä, jotka hyödyntävät järjestelmän rinnakkaisten osien välistä riippumattomuutta. Työssä esitetään kolme vaihtoehtoista osittaisjärjestyssemantiikkaa sekä vertaileva analyysi. Kokeiden perusteella osittaisjärjestyssemantiikoilla voidaan ratkaisevasti nopeuttaa rajoitettua mallintarkastusta. Lisäksi kehitetään rajoitetun mallintarkastuksen muunnelma, jossa lähtökohdaksi on otettu tapahtumien rinnakkaisuus sen sijaan, että tarkasteltaisiin tapahtumia tiukassa aikajärjestyksessä.

Neljäs aihealue on predikaattiabstraktion tehokas laskenta. Predikaattiabstraktiossa luodaan automaattisesti abstrakti malli, josta epäolennaisia yksityiskohtia on jätetty pois. Näin on saatu tarkastettua monimutkaistenkin järjestelmien ominaisuuksia. Käytännössä abstraktin mallin rakentaminen on laskennallisesti raskasta. Tässä työssä kehitetään joukko tekniikoita, jotka hyödyntävät alkuperäisen järjestelmän rakennetta ja paloittelevat abstraktio-ongelman helpompiin osaongelmiin, jolloin kokonaislaskenta-aikaa saadaan pudotettua huomattavastikin.

Avainsanat ohjelmistojen verifiointi, hajautetut järjestelmät, rajoitettu mallintarkastus, UML, predikaattiabstraktio

ISBN (painettu) 978-952-60-4348-7 **ISBN (pdf)** 978-952-60-4349-4

ISSN-L 1799-4934 **ISSN (painettu)** 1799-4934 **ISSN (pdf)** 1799-4942

Julkaisupaikka Espoo **Painopaikka** Helsinki **Vuosi** 2011

Sivumäärä 224 **Luettavissa verkossa osoitteessa** http://lib.tkk.fi/Diss/

Contents

Preface

This thesis is the result of my postgraduate studies and employment as a researcher since 2006 in the Laboratory for Theoretical Computer Science at Helsinki University of Technology, which more recently went through organizational reforms and became part of the Department of Information and Computer Science at Aalto University. My full-time research work has been made possible by financial support from the Helsinki Graduate School in Computer Science and Engineering (Hecse), the Academy of Finland (project 128050), and the SMUML project joint with Tekes (Finnish Funding Agency for Technology and Innovation) and industrial partners (Nokia, Conformiq Software, and Mipro). Moreover, the thesis work has been supported by three personal grants kindly awarded by Jenny and Antti Wihuri Foundation, Emil Aaltonen Foundation, and the Foundation of Nokia Corporation.

I must acknowledge the guidance offered by my supervisor Prof. Ilkka Niemelä, who has great insight and experience on writing scientific text and on the research field. I have enjoyed working with such a skillful leader—I can only imagine the amount of effort Prof. Niemelä has gone through to minimize any financial and practical concerns that his students have to face. Thanks are due to Dr. Tommi Junttila for his uncompromising devotion to the work as my instructor. Besides co-authoring most papers, he has given numerous ideas and feedback on every part of this thesis. I also thank Prof. Keijo Heljanko, who has had an important "mentoring" role in this work as a model checking expert, and I appreciate the encouraging atmosphere and facilities offered by the entire Department staff.

One of the publications in this thesis was prepared during my research visit to Trento, Italy, in 2008. I thank Dr. Alessandro Cimatti, Dr. Marco Roveri, and my other colleagues and friends at the FBK Irst research

center in Trento for the experience that expanded my view on symbolic model checking and research in general.

I am grateful for the vast amounts of peer support from my friends Antti Hyvärinen and Riitta Toivonen, working on their own theses at the same time. My family—parents Yrjö and Marja-Terttu, sister Tanja, and brother Tero—have been supportive all the time and deserve thanks for their understanding.

Finally, I thank the appointed pre-examiners, Prof. Armin Biere and Dr. Stephan Merz, for their expert evaluation of this thesis.

Espoo, October 19, 2011,

Jori Dubrovin

List of Publications

This thesis consists of an overview and of the following publications which are referred to in the text by their Roman numerals.

I Jori Dubrovin and Tommi Junttila. Symbolic Model Checking of Hierarchical UML State Machines. In *Application of Concurrency to System Design, 8th International Conference (ACSD 2008)*, pages 108–117, June 2008.

II Tommi Junttila and Jori Dubrovin. Encoding Queues in Satisfiability Modulo Theories Based Bounded Model Checking. In *Logic for Programming, Artificial Intelligence, and Reasoning, 15th International Conference (LPAR 2008)*, pages 290–304, November 2008.

III Jori Dubrovin and Tommi Junttila and Keijo Heljanko. Symbolic Step Encodings for Object Based Communicating State Machines. In *Formal Methods for Open Object-Based Distributed Systems, 10th IFIP WG 6.1 International Conference (FMOODS 2008)*, pages 96–112, June 2008.

IV Jori Dubrovin and Tommi Junttila and Keijo Heljanko. Exploiting Step Semantics for Efficient Bounded Model Checking of Asynchronous Systems. Accepted for publication in *Science of Computer Programming*, Special Issue on Automated Verification of Critical Systems, 27 pages, available online 23 July 2011.

V Jori Dubrovin. Checking Bounded Reachability in Asynchronous Systems by Symbolic Event Tracing. In *Verification, Model Checking, and*

Abstract Interpretation, 11th International Conference (VMCAI 2010), pages 146–162, January 2010.

VI Alessandro Cimatti and Jori Dubrovin and Tommi Junttila and Marco Roveri. Structure-Aware Computation of Predicate Abstraction. In *Formal Methods in Computer Aided Design, 9th International Conference (FMCAD 2009)*, pages 9–16, November 2009.

Author's Contribution

Publication I: "Symbolic Model Checking of Hierarchical UML State Machines"

The author designed the encoding of hierarchical UML state machines and message queues as a symbolic transition formula. The chosen UML subset and its formal semantics have equivalent contributions from the author and the co-author. The experimental UML model checker was designed and implemented on top of NuSMV by the author. The text was written together.

Publication II: "Encoding Queues in Satisfiability Modulo Theories Based Bounded Model Checking"

Tommi Junttila is the main contributor of this work. The shifting-based queue encoding was designed by the author. The ideas of the cyclic and linear queue encoding and tag-based element compression were originally Tommi Junttila's. Elaborating, analyzing, and writing down these ideas were done in collaboration, and the author implemented them in the context of UML model checking, which contributes to part of the experiments.

Publication III: "Symbolic Step Encodings for Object Based Communicating State Machines"

The idea of employing step semantics was conceived together, but the refinement into the specific types of static and dynamic step semantics was architected by the author. The encoding details were designed and implemented by the author. Keijo Heljanko worked out the relation to earlier

step semantics approaches. A major part of the paper was written by the author, in particular, Sect. 3, as well as most of the other sections.

Publication IV: "Exploiting Step Semantics for Efficient Bounded Model Checking of Asynchronous Systems"

This paper started out as a generalization and elaboration of [**III**]. The policies of which semantics to include and how to apply them were decided together with the co-authors. The design of the abstract system formalism was also collaborative. The definitions and proofs regarding the encoding of actions and the transition formulas are the author's work. The idea of the kind of process semantics used in this paper came from the author, with a linear-size encoding pointed out by Tommi Junttila. The implementation and experiments were conducted by the author. Most of the paper was written by the author. This includes Sects. 3.1–6 almost entirely, and parts of the other sections.

Publication V: "Checking Bounded Reachability in Asynchronous Systems by Symbolic Event Tracing"

The author is the sole contributor.

Publication VI: "Structure-Aware Computation of Predicate Abstraction"

The general concept of structure-aware abstraction is due to Alessandro Cimatti. The final composition of techniques applied to achieve this was designed by Cimatti and the author. In addition, the author contributed the details and the implementation of the low-level abstraction techniques of Sect. IV, and took part in formulating the behavior of hybrid automata networks. The experiments were designed, executed, and reported by the author together with Marco Roveri.

Brief Summary of the Publications

I UML 2.0 state machines offer a semiformal language for modeling the behavior of asynchronous systems. This paper describes a translation that maps a UML state machine to a compact symbolic transition relation formula. The supported UML features include hierarchical state machines, asynchronous messages with data, and deferred events. The translation can be used for checking properties of UML models using various symbolic techniques, such as BDDs or bounded model checking. Furthermore, accurate state space based semantics is given for the chosen UML subset.

II The paper discusses ways to encode the behavior of queues in the context of bounded model checking using SMT techniques. Queues are frequently used to model, for example, message buffers in distributed systems. While modern SMT solvers support some high-level datatypes in their theories, a theory of queues is not directly supported. This work enables SMT-based bounded model checking in the presence of queues, relying only on widely supported theories.

III Bounded model checking of asynchronous systems can be accelerated by exploiting the inherent independence of the components of the system. Using step semantics, several independent actions can be compressed to a single step where the actions are executed in parallel, allowing faster coverage of the reachable state space. This paper shows how to apply step semantics to a class of systems consisting of state machines that communicate through message queues and shared variables.

IV This article extends the ideas of step semantics in several ways and generalizes the handling of data to a more abstract level. The behavior

of systems is represented in a symbolic formalism that covers a variety of modeling languages. Besides a step semantics based on parallel execution of independent actions, a relaxed version of the semantics is presented that achieves even tighter compression of actions into steps. Furthermore, to reduce redundancy in the set of step semantics executions, a variation known as process semantics is defined. Explicit bounded model checking formulas are presented for all semantics and proved for correctness. An extensive benchmark set is used to compare the effects of different execution semantics on the performance of bounded model checking.

V This paper presents a new framework for checking bounded reachability properties using an SMT solver. In contrast to traditional bounded model checking, where an execution is constructed by fixing a sequence of states and considering the possible transitions between them, here we take a fixed set of potential events and let the solver explore the possible flow of control and data between the events. The idea is to even further exploit the independence of components in an asynchronous system. The concept is formalized using a class of high-level Petri nets. Given a Petri net that specifies a bounded portion of the behavior of the system, the paper shows how to automatically construct a formula whose satisfiability corresponds to witnessing the reachability property. The potential of the technique is demonstrated with a proof-of-concept implementation.

VI Counterexample-guided abstraction refinement (CEGAR) is a powerful technique for model checking complex systems, based on disregarding irrelevant details. CEGAR involves repeated computation of an abstraction of the system, essentially by an expensive quantifier elimination procedure. In this work, we exploit the structure of the underlying system to partition the problem into a combination of cheaper abstraction problems. Transformations at the formula level are used to further reduce the scopes of individual quantifiers. Experiments on a set of hybrid system benchmarks are used to evaluate the benefit from the techniques.

1. Introduction

This thesis addresses the problem of design errors in software systems. A desktop PC crashing because of malfunctioning software can make one lose hours of work. A failing controller in a factory may cause expensive downtime. In medical or aerospace equipment, failure might lead to disasters. The more critical tasks we hand over to be carried out using computer-driven systems, the stricter requirements are needed on the correctness of the systems. The challenge here is that these systems are inherently complex. In the real world, systems need to be *reactive*, which means that they must respond to external events on the fly. The systems are *concurrent*, with events occurring in the environment and in different system components at the same time or asynchronously in an order that is hard to predict. The need to deliver rich functionality on a limited hardware platform can force the design to be optimized for performance rather than clarity. All this complexity makes it difficult to predict whether the behavior of the system is acceptable in every possible situation.

While there are many approaches to increase the quality of software systems, such as adopting better design languages and design methodologies, making systems fault-tolerant, and building software from certified components, it is also important to be able to assess the quality. A major part of the assessment is *verification*, which broadly means confirming that a system meets its specification. In software, the prevalent form of verification is testing, i.e. running the system in a real or simulated environment and examining the results. While testing is very efficient in practice, it has a fundamental shortcoming: we can never be sure that all possible behavior is covered by our tests. Especially in the presence of concurrency, the system might fail only after a very specific sequence of events that is very difficult to spot. Even reproducing such a sequence can be a challenge because of fluctuations in the timing of different components.

In *formal verification*, the approach is to use mathematically rigorous reasoning to decide whether a system fulfills its specification. Ideally, this corresponds to guaranteed $100\ \%$ test coverage, and the result is a either a proof that the system is correct or a witness that pinpoints an error in the system design. The justification of correctness can only involve formal reasoning steps without the possibility to rely on human knowledge of the system or its environment. Therefore, one must first construct a *formal model* of the system and represent also the specification formally as a set of desired properties of the model. The system model has to be self-sufficient in the sense that it must contain every detail of the system and every assumption about the operating environment that are needed to show that the specification is fulfilled. Even if formal verification gives an affirmative answer concerning the model and its formal specification, the result is not trustworthy if the model does not reflect the properties of the real system.

Thus, modeling a complex system is a challenge in itself. However, in developing complex embedded systems, the trend is towards *model-based design*, where the model is actually constructed first using a formal or semiformal modeling language, after which the system is implemented based on the model. From the verification point of view, an added benefit is that the design model used for development can also be used for verifying the system, even before the implementation is ready. In the case of pure software components, the implementation itself is formal because it is written in a programming language. This makes it possible to apply formal verification directly to program code, as long as there is a formalization of the semantics of the language. While this work focuses on efficient formal verification, it is not discussed here how to integrate it in the software engineering process. In particular, there is no contribution on how to build the formal system model, how to validate the specification against the intended behavior, or how to repair design errors if they are found.

Writing a mathematical proof that a model of a system is correct with respect to a given formal specification is laborious and often relies on the ingenuity of a verification expert. Parts of the process can be automated using an interactive theorem prover, and this kind of deductive verification has been successfully applied to safety and business critical systems [78, 60]. However, to reduce the cost of formal verification and to turn it to widely adopted push-button technology, fully automatic meth-

ods are a vital objective. A simple idea is to let a computer systematically go through all scenarios that the system can get into and check for violations. This leads to an approach called *model checking* [29]. First, we identify each conceivable snapshot of the system at a point of time as a *state*. The behavior of the system then maps to *transitions* between states. The system model is a description of the states and the transitions, and verification is performed by a tool that analyzes the state-transition graph. All possible behavior of the system is contained in the reachable state space, which is the part of the graph that is reachable by transitions from the initial states.

We cannot hope to write a tool that can automatically verify every piece of software because it is well known that checking non-trivial properties of computer programs is undecidable (Rice's theorem). A manifestation of the undecidability is that the reachable state space of a software system may be infinite. Model checking is sometimes understood to be applicable to only finite-state systems. Infinite-state model checking is viable as well, but generally there is the possibility that the model checker runs forever and does not give an answer. For specific classes of infinite-state models, terminating algorithms have been presented [90, 2].

In the original formulations of model checking [23], the system specification is taken in as a formula in temporal logic. Temporal logics such as LTL and CTL [29] allow describing how events relate to other events in the future or in the past, for example "if a request is sent, then eventually a response will be received". In most cases, the specification we want to check does not mention arbitrary references to the future, such as the word "eventually", but instead has the form "something bad never happens". Such a specification belongs to a class called *safety properties*, whose defining aspect is that the occurrence of "something bad" can be immediately observed. Often the most critical properties of a system are safety properties, for example "the program never attempts to divide by zero" and "the door to the elevator shaft on the 4th floor is never open, unless the elevator is on the 4th floor". The model checking algorithms for safety properties are simpler and more efficient than for the general case, and can be used as a basis for checking more complex temporal properties [89]. The common way to formalize a safety property is to represent it as a desired *state invariant*. Our formal specification is then that every reachable state of the model is safe, that is, fulfills the invariant property.

In this thesis, the main research question is how to efficiently model

check safety properties of concurrent systems with software features. We assume that the system is modeled using a formal language with suitable constructs for concurrency and data manipulation. Model checking is chosen as the verification method because it has proven to cope well with concurrency and requires minimal user interaction. The work thus involves designing model checking algorithms that can handle the high-level data and concurrency features of modeling languages. The primary concern and the most challenging part is the scalability of the algorithms to large systems.

1.1 Basis of the Research

The state explosion problem. Given an invariant property, a set of initial states, and a way to generate the outgoing transitions from a state, model checking reduces to the problem of reachability in the state-transition graph. If the graph is finite, the algorithm that solves this is elementary. *Explicit-state model checking* refers to methods that search through the states one by one, and an efficient implementation covers millions of states in seconds. Unfortunately, this is not enough in practice. A model of a concurrent system of quite modest complexity can have a few million reachable states or less, but the number grows rapidly when complexity increases. This *state explosion* is the major challenge in model checking.

State explosion is caused by *nondeterminism* in the model, which means that states may have several outgoing transitions. There are two main sources of nondeterminism. One is input from the environment. Reactivity means that new input may be received in any state, and every possible input generates a new transition leaving from that state. Especially if the system receives data, every data value leads to a different state. The other source is concurrency. The execution platform and the implementation can seldom be modeled accurately enough to predict the execution speeds of asynchronous concurrent components. As a conservative approximation, the order of execution is nondeterministic in the model. The crudest and most common approximation is the *interleaving* model of concurrency, where in each state, one component is nondeterministically chosen to execute one atomic action while the other components remain still. Increasing the number of concurrent components or data variables

causes a combinatorial explosion in the number of states. Generally, the reachable state space grows much faster than the size of the model, exponentially in the worst case.

Symbolic model checking. Evidently, searching through all reachable states does not scale up. In *symbolic model checking*, the idea of enumerating states is dismissed. Instead of manipulating states and transitions individually, they are manipulated in sets that are represented in a symbolic form. For example, if an integer value $s \in \mathbb{Z}$ is viewed as a state and a pair $(s, s') \in \mathbb{Z}^2$ as a transition from s to s', then the formula $s < s' \leq 10$ represents symbolically the set of transitions where s is incremented nondeterministically, but not beyond the value 10. The languages used for the representation are based on Boolean logic, which allows using efficient tools of computational logic for the symbolic manipulation.

The symbolic method that was introduced first is to encode the states as fixed-length Boolean vectors and to use reduced ordered binary decision diagrams (*BDDs*) to express the Boolean functions that correspond to sets of states and transitions [17]. BDDs allow a compact representation of large state spaces especially in the case of synchronous hardware designs with a regular structure. However, not all Boolean functions have a succinct BDD, and model checking often fails because the BDD for the set of reached states becomes too large.

Bounded model checking (BMC) [11] leverages the rapid evolution of propositional satisfiability (SAT) solvers and avoids the memory issue of BDDs by taking an even more implicit view on the reachable state space. Using the symbolic representation of transitions, it is straightforward to construct a formula that characterizes all executions of at most a fixed length k that lead from an initial state to any state that violates the invariant property. The formula is satisfiable if and only if such an execution exists. To decide whether the property can be violated within k execution steps, the satisfiability of the constructed formula is checked using a SAT solver. If the formula is found to be unsatisfiable, then typically the check is done again with the bound $k + 1$, and so on. Unlike with BDDs, there is no easy way to determine when all reachable states have been covered. Proposals have been made to compute a bound k that guarantees complete coverage [62, 40], or otherwise make BMC complete for finite-state systems [91, 68, 71]. Even as an incomplete verification

method, BMC has a lot of practical value because of its ability to quickly find shallow errors that can be witnessed by a short execution [82, 4, 30].

During the last two decades, symbolic model checking has become a routine in hardware verification [63]. Generally, verification of software is considered more difficult than hardware because software has less regular structure and contains features such as recursion, pointers, and data types with potentially unbounded domains. In this work, we are adapting symbolic methods to handle concurrent software systems. The expected benefit of employing symbolic model checking is that it copes better with nondeterminism than explicit-state model checking. Especially when nondeterminism is combined with a large data domain typical of software, symbolic methods can avoid separately examining each data value. In some approaches to model checking sequential software, the control flow graph is represented explicitly and control locations are associated with symbolic sets of the values of variables [50, 70, 79]. In a concurrent system however, the control flow is more convoluted because there is no single point of control. Under the interleaving model of concurrency, any component can be scheduled for execution at any time, and a state of the system model tells the current active control location of each component as well as the values of data variables. To apply BDD-based and bounded model checking, we encode all this in a symbolic form. Another possibility would be to start with a sequential model of each component and glue them together by adding nondeterministic context switches between components [83, 41]. The work on pushdown systems [90] aims at model checking software with unbounded recursion, and it has been extended to multithreading with bounded context switches [95]. The approach chosen in this thesis only supports bounded recursion via inlining function calls.

Partial order methods. The high number of interleavings of a concurrent model is a cause of state explosion, and *partial order methods* [44] have been introduced to diminish the effect. In explicit-state model checking, this is implemented in a form of state space reduction, where the idea is to exploit regularities of the state space to omit parts of it without affecting the property to check. Partial order reduction is based on the observation that the nondeterminism introduced by different interleavings is often redundant. If several components are about to execute an action and the actions are pairwise independent, the same state is reached re-

gardless of the order in which the actions are executed. Reducing equivalent interleavings to a single representative can greatly reduce the search space and speed up explicit-state model checking [44]. For symbolic model checking, there are partial order methods based on the same idea (see Sect. 5.4), although the approach is different from state space reduction because the cost of symbolic model checking does not correlate directly with the number of reachable states. Symbolic partial order methods can significantly accelerate the BMC of concurrent systems and are a major topic of this thesis.

Satisfiability modulo theories. In the software domain, a potential disadvantage of symbolic model checking is the cumbersome mapping of software features to Boolean formulas. For example, if a transition involves adding two 32-bit integers, then a 32-bit binary addition circuit needs to be encoded in the formula that represents the transitions. Increasing the instances of subcircuits like this can overpower the SAT solver or the BDD engine. The need to reduce the bulky low-level encoding of data features has motivated the development of satisfiability solvers that can check more expressive formulas than pure propositional logic. *Satisfiability modulo theories* (SMT) [10] combines Boolean logic with constraints expressed in background theories, such as the theory of linear arithmetic, fixed-length bitvectors, or unbounded arrays. In bounded model checking, switching the back-end from a SAT solver to an SMT solver means that model elements such as integer variables and pointers no longer need to be broken down to individual Boolean variables but can be passed to the solver as higher level constructs [6]. The ongoing evolution of modern SMT solvers has a positive effect in model checking performance. On the other hand, the rich language support opens up many possibilities of expressing problems, and in many cases it is not clear which kind of logic encoding gives the best results. Finding efficient encodings is a concern also in this thesis, much of which relies on heavy use of SAT and SMT solvers.

Abstraction. A key technique employed in recent successful formal verification efforts [7, 50, 28] is *abstraction*, which means simplifying the problem by disregarding irrelevant details. The formal model is already a manual abstraction of the system, but in model checking, the idea is used

in a systematic way to circumvent state explosion. In *existential abstraction* [25], an abstract model is constructed automatically in a conservative way that over-approximates the behavior of the concrete model. To verify a safety property of the concrete model, it is sufficient to show that the simpler abstract model fulfills the property. The abstraction generally introduces nondeterminism, so a symbolic method such as BDDs is typically used to check the abstract model. If the abstract model is too coarse, it does not contain enough details for verification, but including too many details brings back the complexity of the concrete model. Several abstraction refinement schemes have been proposed to iteratively find a sufficient level of detail [24, 71]. A prevalent form of existential abstraction is *predicate abstraction* [45], which is attractive for software verification because it maps even an infinite-state model to a finite abstract model whose state is a Boolean vector. Given a number of state predicates that evaluate to true in some states and to false in the rest, those concrete states that give the same values to all state predicates are collapsed into a single abstract state. While predicate abstraction is effective in reducing the state space, the bottleneck of model checking can shift from searching the state space to constructing the abstract model from the concrete model. The expense of computing the predicate abstraction is one of the problems addressed in this work.

1.2 Contributions of the Thesis

The contributions of this thesis are divided to four topics. The topics are briefly summarized below, and they are further explained in Sects. 3–6, respectively. First, general background information about the model checking techniques is given in Sect. 2.

1. *Symbolic model checking of UML state machine models* [**I**].

   UML is a modeling language for software-intensive systems, and its wide industrial use motivates verification techniques that take UML models as input. UML 2.0 state machines offer a semiformal language for modeling the behavior of asynchronous systems. The contribution is an accurate semantics for a subset of the UML state machine language, defining the state space of the model, and a translation that maps the model to a compact symbolic transition formula. The supported UML

features include hierarchical state machines, asynchronous messages with data, and deferred events. The translation can be used for checking properties of UML models using symbolic techniques based on BDDs or bounded model checking.

2. *Bounded model checking of systems with queues* [**I**], [**II**].

   First-in-first-out queues are frequently used to model, for example, message buffers in distributed systems. While modern SMT solvers support some high-level datatypes in their theories, the theory of queues is not directly supported. The contribution is a variety of ways to encode the behavior of queues in SMT-based bounded model checking, relying only on widely supported theories.

3. *Symbolic partial order methods for accelerated model checking* [**III**], [**IV**], [**V**].

   To speed up bounded model checking of concurrent systems, we generalize the interleaving execution semantics to so-called step semantics, which allow several independent actions to be executed in a single step. This brings down the required number of execution steps to find a property violation, which is a critical aspect regarding the performance of bounded model checking. This work presents two practical variants of step semantics, called the parallel $\exists$-step semantics and the serial $\exists$-step semantics. While their ideas have been introduced earlier, here the two step semantics are adapted to systems with data variables of unrestricted domains. Also, the notion of independence is generalized to allow nondeterministic and context-sensitive dependencies between actions. Furthermore, a novel semantics, called the serial process semantics, is designed as a normalized form of the serial $\exists$-step semantics. It is shown that the serial process semantics corresponds one-to-one to the partial order semantics of Mazurkiewicz traces and thus eliminates the redundancy caused by the mutual ordering of independent actions. The three semantics, as well as the interleaving semantics, are presented in an abstract, unified framework and compared to each other analytically and experimentally on an extensive benchmark set.

   Another contribution is a new technique for reducing the bounded reachability problem in a concurrent system to an SMT problem. Unlike traditional bounded model checking, this technique, called bounded event tracing, is inherently a partial order method. Instead of tying

the executed actions to fixed global time steps, they are locally linked to other actions through the flow of control or data. The (partial) order of execution is then implied by these links. This idea is presented as a generic framework based on high-level Petri nets with bounded execution semantics and automatic generation of corresponding SMT formulas. A translation from a class of state machine models to this framework is defined and experimentally tested. However, there is not yet a translation scheme that is efficient in the general case.

4. *Efficient computation of predicate abstraction* [**VI**].

In predicate abstraction, computing the transitions of the abstract model can be more expensive than model checking the abstract model. Earlier approaches compute a symbolic representation of the abstract transitions in a monolithic way. In this work, we instead build the set of abstract transitions from parts by following the structure of the system model. A number of techniques is represented for partitioning the problem, and each part is solved as a local abstraction problem. As the computation of the abstraction is essentially a quantifier elimination problem, and the local problems contain fewer variables under the scope of the quantifier than the monolithic problem, this can speed up the computation remarkably. The idea is instantiated for systems expressed as networks of linear hybrid automata. Experiments on a set of benchmarks demonstrate the benefit from the techniques.

2. Background

In this section, we will briefly review satisfiability solving (Sect. 2.1), symbolic model checking in general (Sect. 2.2), bounded model checking (Sect. 2.2.1), and a generic concurrent system model (Sect. 2.3) and its symbolic transition formula (Sect. 2.3.1). These concepts and definitions will be used as a baseline throughout the rest of the thesis.

2.1 Satisfiability Modulo Theories

Several of our techniques rely on tools that decide the satisfiability of quantifier-free formulas in first-order logic. Given a well-formed formula constructed from function and predicate symbols and Boolean connectives, the formula is satisfiable iff there is an interpretation of the symbols that evaluates the formula to true. More precisely, we are interested in satisfiability modulo theories (SMT) [10]. This means that we only take into account interpretations that respect one or more background theories, which fix the interpretations of some symbols. An example of such a theory is linear integer arithmetic, which fixes the interpretation of the symbols $<$, $+$, 0, 1, and so on.

Other theories relevant to this work include the theory of fixed-length bitvectors, the theory of difference logic, and the theory of arrays. Difference logic encompasses theory atoms of the forms $x - y = c$ and $x - y \leq c$, where x and y are variables and c is an interpreted real constant. (We say "variable" to refer to an uninterpreted nullary function or predicate symbol.) The theory of arrays defines function symbols $read$ and $write$ of arities 2 and 3, respectively. The function application $read(a, i)$ evaluates to the value at index i in the array a. The meaning of $write(a, i, v)$ is an array identical to a except that the value at index i is v. As an example of

using an uninterpreted function (uif), the formula $(f(x) \neq f(y)) \wedge (x = y)$ is unsatisfiable: no interpretation of the function symbol f can make the formula true.

In this thesis, we employ modern SMT solver tools, which are capable of checking satisfiability modulo these and other theories and also theory combinations [10, 93]. In many cases, the formulas can be constructed using only Boolean variables and Boolean connectives, and their satisfiability can be checked using a propositional satisfiability (SAT) solver.

2.2 Symbolic Model Checking

In this work, the term symbolic model checking is used to refer to checking invariant properties using BDD-based image computation or satisfiability-based bounded model checking. BDDs (reduced ordered binary decision diagrams [15]) are a canonical representation for Boolean functions, and they can be manipulated by efficient algorithms.

We assume that the system model defines a finite set of *state variables*, and we identify a state with a vector that gives values to the state variables. The initial states are defined by a predicate I such that a state s is initial if and only if $I(s)$ is true. The transitions of the state space are defined by a given predicate T on pairs of states: there is a transition from a state s to s' if and only if $T(s, s')$ is true. The property to check is whether every state reachable by transitions from any initial state satisfies an invariant property, also given as a state predicate P.

BDD-based symbolic model checking [29], in its simplest form, is a direct implementation of the following idea. We define state predicates $S_0, S_1, S_2, \ldots$ by

$$S_0(s) := I(s) \tag{2.1}$$

$$S_{t+1}(s) := S_t(s) \vee \exists s'' : S_t(s'') \wedge T(s'', s) \qquad \text{for } t = 0, 1, \ldots. \tag{2.2}$$

In words, S_{t+1} is formed by adding to S_t all states reachable by a transition from S_t. Thus, S_t is true for exactly those states reachable from an initial state by a sequence of t or fewer transitions. If the equivalence $S_{t+1} \equiv S_t$ holds for some t, then the recursive definition has reached a fixpoint, and S_t characterizes precisely the reachable states. If, in addition, $S_t \wedge \neg P$ is unsatisfiable, then the invariant property holds in the model. On the other hand, if there is any t such that $S_t \wedge \neg P$ is satisfi-

able, then any satisfying interpretation constitutes a state that violates the invariant property and is reachable in at most t steps from an initial state. Then, it is straightforward to extract an execution path that acts as a counterexample to the property [29].

To apply BDDs, the state variables need to be Boolean or at least have a finite domain with an implicit assumption of a mapping to Boolean vectors. The initial predicate I and transition predicate T are encoded as BDDs, and the forward image computation (2.2) is iterated until either a fixpoint is reached or a property violation is found. As BDDs are canonical, the equivalence and satisfiability checks are trivial. Because of the finite state space, this process always terminates. The practical challenges are the expensive quantifier elimination in (2.2) and the fact that the BDD representations of the intermediate reachability predicates S_t tend to blow up in size.

2.2.1 Bounded Model Checking

Bounded model checking (BMC [11]) is a symbolic technique that avoids constructing representations of sets of reachable states and thus circumvents some of the bottlenecks of BDDs. In this work, we treat BMC as an incomplete method for falsifying invariant properties. Direct contributions are not made on complete BMC (proving invariant properties) nor on more general temporal properties.

Given a bound $k \geq 0$, a sequence of states $s_0, \ldots, s_k$ that satisfies

$$I(s_0) \wedge \bigwedge_{t=0}^{k-1} T(s_t, s_{t+1}) \wedge \neg P(s_k) \tag{2.3}$$

constitutes a counterexample that breaks the invariant property P. In BMC, we encode the predicates I, T, and P as formulas and instantiate them over the $k+1$ timed copies of the vector s of state variables to construct (2.3) as a formula. In particular, the *transition formula* T is unrolled k times. The satisfiability of the BMC formula (2.3) is checked iteratively for bounds $k = 0, 1, 2, \ldots$. The formulas need to be encoded in propositional logic for satisfiability checking with a SAT solver, or in a supported quantifier-free theory for an SMT solver. If a satisfying interpretation is found, it directly gives a counterexample to the property. If the formula is unsatisfiable up to bound k, we know that every state reachable in k or fewer steps respects the invariant property, and the check is repeated with an increased bound. If every reachable state sat-

isfies the property, this process does not terminate. In practice, the bottleneck is that the satisfiability solving time grows quickly, even exponentially, when the bound is increased. One way to improve this is to use incremental satisfiability checking [35], which avoids solving each BMC instance from the ground up.

Although the only variables in (2.3) are the timed state variables, we allow the transition formula $T(s, s')$ to contain auxiliary variables that facilitate the encoding. Essentially, the transition formula can have the form $T(s, s') \equiv \exists c : \widetilde{T}(s, c, s')$, and the satisfiability checking automatically takes care of existentially quantifying the vector of encoding variables c.

We note that although other formulations of BMC exist (e.g. [27]), the term BMC in this work refers to the formulation (2.3) based on unrolling a transition formula.

2.3 A Generic Concurrent System Model

As a generic model of concurrent systems with discrete, asynchronous execution steps, we consider a model whose states are defined by a finite set of *state variables* and whose behavior is defined by a finite set of *actions*. Each action is associated with a subset of states in which the action is *enabled* for execution. When an action is executed, its *effect* is to assign—possibly nondeterministically—new values to some of the state variables. The model follows the interleaving execution semantics: one atomic execution step corresponds to nondeterministically choosing an enabled action and executing it. A formal treatment of this generic model is given in [**IV**].

In this model, the division of the system into components is not explicit, nor is the control flow of the components. However, the underlying idea is that the state variables can represent either control locations or local data of components or shared data, and an action can represent a local operation within a component, or it can model synchronization or communication between components. For example, the actions might correspond to the transitions of a UML state machine model, with a state variable allocated for each state machine to hold the current active state.

2.3.1 Transition Formulas

For a concrete understanding of how we intend to apply symbolic model checking, we briefly overview two ways to formulate a transition formula for models described above. The first way is to write a transition formula whose structure is a disjunction over the actions, numbered from 1 to n. The general form is

$$T_{disj} \equiv \bigvee_{i=1}^{n} (enabled^i \wedge effect^i \wedge frame^i). \tag{2.4}$$

Above, each subformula $enabled^i$ tells whether the ith action is enabled in the current state s. This typically includes at least checking the current control location of the corresponding component. The subformulas $effect^i$ constrain the next-state variables s' to respect the changes made by the action, for example, setting the new active control location. The frame conditions $frame^i$ ensure that the remaining next-state variables retain their current values.

A potential source of inefficiency in the formulation (2.4) is the amount of repetition in the frame conditions. In particular, each state variable local to a component has to be mentioned in the frame condition of every local action of every other component. Our second formulation reduces such coupling.

For the alternative transition formula, we introduce a set of auxiliary Boolean variables $f^1, \ldots, f^n$. Their meaning is that f^i is true iff action i is the one being executed. We factor out the frame conditions into a common formula $frame$, which has a compact representation based on the f^i variables and information on which state variables each action changes. Using these elements, we rewrite the transition formula in a conjunctive form that can be generally expressed as

$$\begin{aligned} T_{conj} \equiv \bigwedge_{i=1}^{n} (f^i \rightarrow enabled^i) \wedge \bigwedge_{i=1}^{n} (f^i \rightarrow effect^i) \wedge \\ frame \wedge one\text{-}hot(f^1, \ldots, f^n). \end{aligned} \tag{2.5}$$

Above, the $one\text{-}hot$ constraint makes sure that exactly one of the f^i variables is true.

For a detailed construction of a conjunctive transition formula, see the description of the interleaving transition formula in [**IV**]. In Sect. 5, this is used as a baseline for transition formulas that follow alternative partial order semantics. Also, the transition formula for UML models (Sect. 3)

follows the conjunctive structure; however, the formula is optimized by factoring out common logic shared by actions in the same state machine. The partitioning of the predicate abstraction problem in Sect. 6 is based on a disjunctive representation like (2.4). The queue encodings of Sect. 4 can be combined with a transition formula of either shape.

3. Symbolic Model Checking of UML State Machines

In systems development, new technology that plugs in transparently is accepted much more readily than technology that disrupts the development flow [63]. Requiring the developer to change the design language would be a major disruption. Therefore, model checkers that support current and emerging industrial languages are at a key position in paving the way for widely adopted model checking. As the popular languages are not designed for verifiability, this poses technical challenges. One is the lack of formal semantics. It is not easy to precisely characterize features such as pointer casting in C, reflection in Java, or calls to library functions that may be underspecified. Another issue is how the semantics maps to the underlying verification technology. Often, this is realized as a translation to the input language of an existing model checker, or in the case of symbolic model checking, to a transition formula. Publication [**I**] addresses these issues for the language of UML state machine models, contributing a semantics and a translation to a symbolic transition formula.

UML or Unified Modeling Language [76] is a dominant language for modeling software systems in academia and industry. Some of its key aspects are object orientation, graphical notations for visual modeling and communication, independence from implementation languages, and orientation to large distributed systems. UML is also a big and complex language. Besides aspects of object oriented programming such as classes, polymorphism, and encapsulation, UML can express business processes, use cases, and deployments of hardware components among other things. From verification point of view, the most interesting features are the behavioral model elements such as state machines for modeling discrete event-driven behavior, sequence diagrams that specify interactions between components, and activities that work like extended flowcharts. As the usage of UML extends from visual documentation to design, simula-

tion, and code generation, there is a rising need to verify UML models.

3.1 UML Subset

A user of UML needs to decide which parts of the system to include in the model, and which UML elements to use for modeling. In [**I**], we choose a language subset based on communicating UML state machines.

In our setting, a run-time instance of a UML model consists of a set of *active objects*. These are objects that not only encapsulate data and behavior but can themselves initiate behavior concurrently with the rest of the system. In programming language terms, every active object has its own thread. We assume that the behavior of each active object is governed by a state machine.

Objects communicate asynchronously by sending *messages* to each other. Each object has an *event pool* from which it dispatches events and reacts to them as specified by its state machine. The only kind of event we consider is message reception. UML does not specify the order in which events are dispatched, but for practicality and simplicity, we assume first-in-first-out order. Thus, the event pool is a queue of messages, and sending a message means placing the message at the end of the queue of the receiving object.

Objects have *attributes*, i.e. instance variables, which may include references to objects. The formalism does not preclude reading and writing the attributes of other objects. This allows communication through shared memory as well as message passing.

State machines consist of *state vertices* and *transitions* between them. A transition can fire if its source state vertex is active, its *guard* condition (a Boolean expression) is satisfied, and an event is dispatched that matches the *trigger* of the transition. So-called *completion transitions* have no trigger and fire spontaneously. In UML, if a dispatched event does not enable firing any transitions, it is lost by *implicit consumption*. However, if the event is marked *deferrable* by an active state, it is saved in a collection of deferred events and will be dispatched again later.

A transition can specify an *effect* that is executed upon firing. UML does not specify an action language for guards and effects, and we also do not fix the language in [**I**]. The model checker implementation uses the action language described in [**III**], which has statements for manipulating 32-bit integer attributes and sending messages with data parameters.

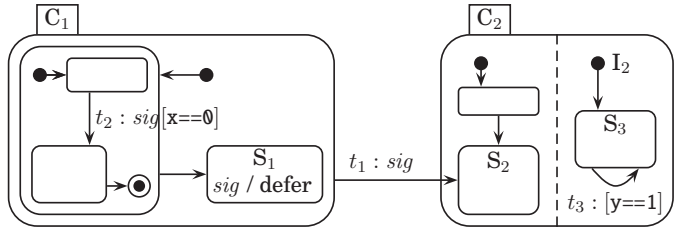


Figure 3.1. A hierarchical UML state machine.

UML allows state machines to be hierarchical, as depicted in Fig. 3.1. A *composite state* is a container for child state vertices, which can in turn be composite states. This feature facilitates top-down design, allowing the behavior of a state to be refined by adding children within it. Hierarchy also improves expressiveness. For example, the single high-level transition t_1 in Fig. 3.1 substitutes a whole set of transitions leaving the descendant states of C_1. The children of a composite state, such as C_2 in the figure, can be divided into two or more *orthogonal regions*. When the composite state is entered, all its regions become active and start functioning like concurrent state machines. Therefore, besides concurrency between active objects, the model allows concurrency within an object.

The other kinds of state vertices in our subset are *initial pseudostates* and *final states*, which mark the start and the end of the behavior of a region or the entire state machine, and *choice pseudostates* that are used as control flow branches.

3.2 UML Semantics

UML is more predominantly a communication tool between people than a machine-executable language. The specification [76] gives an informal account of how the behavioral elements work, but the syntax allows corner cases whose meaning is ambiguous [38]. UML is also intentionally underspecified, with many "semantic variation points" that leave open the choice of semantics. Nevertheless, any tool that performs simulation or verification of UML models must either follow a precise and explicit semantics, or worse, define a semantics implicitly.

In [**I**], we define a semantics for the UML subset described above. The semantics is expressed in an operational style in terms of the state space of a given UML model. In the UML context, we call a vertex of the state

space a *global configuration* to avoid a name clash with a "state" in a state machine. In short, a global configuration tells which state vertices are active in the state machines, what are the attribute values of objects, and what are the contents of the event pools of objects. A single execution step corresponds to a state machine firing one enabled transition or dispatching an event that does not enable a transition. We define explicitly the set of legal execution steps between global configurations. This facilitates the construction of a symbolic transition formula because the two representations are fairly close in structure.

The UML behavior is based on asynchronously executing active objects, but UML by itself does not enforce a scheduling policy. Because we allow active objects to share variables, it is relevant at which points an object can preempt the execution of another object. Our semantics allows arbitrary interleavings with the assumption that each individual transition is executed atomically.

The simplest finite state machine based languages have well understood semantics, but UML adds intricate features such as state hierarchy, deferrable events, and completion transitions, whose combined behavior needs to be included in the formalization. In the presence of composite states, transitions at different levels of the hierarchy can be simultaneously enabled by the same dispatched event. According to the UML specification, transitions whose source state is deeper in the hierarchy have a priority. A particular aspect of UML state machines is *run-to-completion* semantics. This means that a state machine does not dispatch a new event from its event pool until it has completely processed the previous event. The formal semantics of [**I**] takes specially into account the combination of run-to-completion and the intended behavior of completion transitions. For example, the completion transition t_3 in Fig. 3.1 is triggered by an implicit *completion event* that is generated when the source state S_3 becomes active. However, if the guard y==1 is false, the transition is not fired and the run-to-completion step ends. The UML specification implies that even if y==1 becomes true later by an assignment in an orthogonal region or another active object, transition t_3 will not be triggered again. Therefore, we need to subdivide the global configurations where S_3 is active into those where the completion event has not yet been dispatched and those where it has. In the latter case, we say that state S_3 is *quiescent*. This distinction is often overlooked, which may result in both spurious firing of completion transitions and missing legal behavior because of a spurious completion

transition overriding another transition.

3.3 Encoding State Machine Behavior

To enable using BDD-based and bounded model checking as described in Sect. 2.2.1, the state space needs to be represented in a symbolic form. This includes the state variables for encoding global configurations, the initial state formula, and the transition formula. Of these, the transition formula is the most significant part. The initial state formula is not discussed as it is straightforward to construct.

The state variables include (i) Boolean variables for each state vertex in each state machine denoting whether the state vertex is inactive, active, or active and quiescent. (ii) a variable for each attribute of each object, and (iii) variables for the messages in the event pools. Attributes and messages are represented as vectors of Boolean variables for BDD and SAT based model checking, or as variables of a richer sort if an SMT back-end is used. To keep the symbolic representation finite, there is a fixed maximum number of messages that each event pool can contain. Parts of the state space where the limit is exceeded are not covered in the verification.

The transition formula can be divided into three loosely connected parts as follows.

- The dynamics of message queues that constitute the event pools. This is discussed in Sect. 4, where different queue encoding approaches are examined. Any of the approaches can be used to encode the event pools of UML objects. However, the shifting approach described in [**I**] is the only one that supports deferring of events.

- The control logic of state machines, briefly discussed below. The detailed description is in [**I**].

- The guards and effects of transitions. The formulas that encode the guards and effects are assumed to be given and not defined in [**I**]. They depend on the action language, which is not specified. However, one possible implementation is described in [**III**].

The encoding of control logic follows the semantics of state machines dis-

cussed in the previous section. One design goal is to make the encoding compact in terms of formula size for efficient BMC. The factors that affect the run time of a SAT or SMT solver are difficult to estimate, but as a heuristic, we try to keep the size of the input formula small.

Much of the potential complexity of the control logic formula comes from state hierarchy. Consider transition t_1 in Fig. 3.1. The transition is fired when the event sig is dispatched. However, according to the UML transition firing priorities, t_1 is not fired if transition t_2 is also enabled, nor if state S_1 is active because it defers the event. Furthermore, upon firing t_1, not only the target state S_2 becomes active but also the initial pseudostate I_2. In the worst case, there is a quadratic number of implicit dependencies between the transitions and the state vertices of a state machine. In our encoding, these dependencies are structured in a way that follows the state hierarchy. For example, the firing of t_1 is not conditioned directly on state S_1, but on an auxiliary formula that tells whether the source state C_1 has a descendant that defers the event sig. By this construction, the size of the resulting formula is linear in the number of model elements in the state machine.

Publication [**I**] presents a purely mechanical translation of UML models to transition formulas, and the translation has been implemented in the toolset developed in the SMUML project [92].

3.4 Related Work

Foundational UML [75] is the official UML semantics proposal by the Object Management Group (OMG), who also manage the UML standard. A related UML 2 Semantics Project has been carried out by academic and industrial partners [13, 32, 33]. The idea is to define a virtual machine that executes models in a foundational subset of UML, which contains the most basic behavioral elements. In particular, state machines are not part of the Foundational UML. The intent is to specify higher-level elements such as state machines by mapping them to the Foundational UML, but the OMG has not published a realization of this.

Regarding UML state machine semantics, perhaps the most complete account is given in [37]. That work describes a formal semantics for UML-like “core state machines” and a translation from UML to core state machines. The treatment includes many UML features that are not handled

in [**I**], such as history, entry, and exit pseudostates. The granularity of the semantics is much finer in [37], which means going through possibly several intermediate configurations when firing a transition. Using such small steps as such in model checking might increase state explosion and degrade performance. To the present author's knowledge, there are no verification tools based on the semantics of [37].

In [84], the output of UML-to-Java code generators is verified against the semantics of UML state machines. The work employs a partial specification of the semantics derived from the UML standard, covering only selected aspects of the behavior. Other efforts to formalize the semantics of UML state machines are summarized in [88].

For verifying safety properties of UML models, a recent work by Hansen et al. [47] sketches a translation from a subset of executable UML to the mCRL2 language, which has both explicit-state and symbolic model checking back-ends. The UML subset is similar to ours, including hierarchical state machines and bounded event queues but no data attributes. Earlier work on UML model checking include explicit-state [87, 57] and symbolic approaches—see the related work section in [**I**].

3.5 Discussion

Publication [**I**] presents an execution semantics and a transition formula that enable symbolic model checking of UML state machine models. The transition formula is linear-size with respect to the UML description even in the presence of hierarchical state machines. This continues previous work by the same group [57], where a UML semantics and a translation for explicit-state model checking were presented. The semantics in [57] is compatible with [**I**] but in a presentation style that better matches explicit-state model checking.

The transition formula for UML state machines is tested with experiments in [**I**] and also in [**III**]—the latter only concerns flat state machines without state hierarchy. The results suggest that for hierarchical UML state machines, constructing a compact transition formula based on the hierarchy as in [**I**] may perform better than a generic transition formula that requires explicitly flattening the state machines. In comparison to explicit-state model checking, neither our BDD- nor BMC-based implementations generally reach the performance level of the (highly op-

timized) explicit-state model checker Spin [51] on the tested models. Nevertheless, the ability to construct symbolic representations of the behavior of UML models as demonstrated in [**I**] is a step towards applying more advanced symbolic techniques to industrial UML designs.

4. Queue Encodings for Bounded Model Checking

In our domain of systems with no global synchronization, a common communication mechanism is asynchronous message passing, where a component sends a message and does not stop to wait until the recipient has handled the message. In the meantime, the message is kept in a buffer, possibly with other messages waiting to be processed. Since we want to catch the possible errors stemming from unexpected concurrent interactions, we need to incorporate the asynchronous communication in the system model, even if the model presents the internals of components at an abstract level. Often, communication buffers constitute the most complex data structure in a model. If the messages are processed in first-in-first-out order, then we can model the communication buffer as a queue of messages. Explicit-state model checkers such as Spin [51] and DiVinE [9] support such buffers natively in their input languages. In this section, we discuss ways to handle queues in the context of symbolic model checking, as presented in publications [**I**] and [**II**]. We mostly consider bounded queues with a fixed maximum number of elements.

Instead of a queue, one could use an unordered collection (a multiset) to model a buffer from which messages are removed in an arbitrary order. We do not consider this possibility further in this work. Nevertheless, it is sometimes desirable to deviate from strict first-in-first-out order of processing messages. For example, Spin has a "random receive" capability that allows removing messages from the middle of a queue. We take a similar approach and extend the semantics of queues with a *defer* operation, originating from the UML state machine language (Sect. 3.1).

The contribution of this section is a set of alternative ways to encode the behavior of queues in the context of symbolic model checking, in particular, SMT-based bounded model checking. Although the core idea of SMT is to offer decision procedures for theories of high-level data types, a theory

of queues (such as one presented in [12]) is not directly supported in the SMT-LIB format nor in the current SMT solvers. Our queue encodings are based on widely supported theories such as arrays and uninterpreted functions. Some encodings only rely on Boolean and bitvector variables and can thus be used in SAT or BDD-based as well as SMT-based model checking. Publication [**I**] presents a queue encoding as part of the UML system transition formula. Publication [**II**] extends this into a variety of queue encodings, abstracting from the rest of the transition formula. However, the defer operation is only supported in the encoding of [**I**].

We divide the evolution of the contents of a queue into steps that correspond to the BMC time steps. The set of operations that can be applied in a single step is limited—basically, at most one element can be removed from the queue and at most one can be added. This design choice avoids introducing new variables to hold the intermediate queue contents within a step, while still allowing us to model an action that first receives (dequeues) a message and then, in the same step, sends (enqueues) another message, possibly to the same queue. In particular, such an action matches the granularity of execution steps in our UML semantics in Sect. 3.2. Although we are primarily interested in queues as communication buffers, the interface allows using queues for any data that can be compared for equality in SMT.

In Sect. 4.1 below, we will define a common interface that encapsulates the different queue encodings and exposes the allowed queue operations at each time step. The semantics of queues is also described. The encodings are presented in Sect. 4.2, with related work in Sect. 4.3 and a summary and discussion in Sect. 4.4.

4.1 The Queue Interface

We encapsulate the queue encodings behind an interface that consists of expressions for accessing the queue contents and client-controlled variables that determine the operations on the queue. Here, the client is a BMC transition formula unrolled up to a bound k. In the following, we describe an interface that covers the first-in-first-out queue operations of [**II**] and also the deferring features of [**I**]. For the presentation, we take the notation of [**II**] and extend it as needed.

Consider a queue over elements of type ELEM that has a fixed upper

bound Z on the number of elements or is unbounded with $Z = \infty$. We represent the contents of the queue as a pair

$$Q = \langle \langle d_1, \ldots, d_m \rangle, \langle p_1, \ldots, p_n \rangle \rangle,$$

where $d_1, \ldots, d_m \in \text{ELEM}$ are *deferred elements* and $p_1, \ldots, p_n \in \text{ELEM}$ are *pending elements*. The numbers m and n can be zero or positive, and the sum $m + n$ must not exceed the queue capacity Z. The usual case is that there are no deferred elements, and the queue functions in the familiar first-in-first-out basis on the pending elements.

The queue interface consists of expressions for accessing the queue contents and client-controlled variables that determine the operations on the queue. We will denote the queue contents at different BMC time steps by $Q^0, Q^1, \ldots, Q^k$. For each time step t with $0 \leq t \leq k$, the queue interface exposes the following accessor expressions.

- The Boolean formulas empty^t, $\mathit{pending}^t$, and full^t tell whether the queue at step t is empty, has at least one pending element, or is full, respectively. A bounded queue is full iff it contains Z elements, and an unbounded queue is never full.

- The expression $\mathit{firstelem}^t$ of type ELEM holds the value of the first pending element in Q^t. It has a meaningless value if the queue contains no pending elements.

- For $0 \leq u \leq k$, the Boolean formula $\mathit{equal}^{t,u}$ is true iff the contents of the queue at time steps t and u are the same.

Denoting $Q^t = \langle \langle d_1, \ldots, d_m \rangle, \langle p_1, \ldots, p_n \rangle \rangle$, the semantics of the accessors is as follows.

$$\begin{aligned} \mathit{empty}^t &= (m + n = 0), \\ \mathit{pending}^t &= (n > 0), \\ \mathit{full}^t &= (m + n = Z), \\ \mathit{firstelem}^t &= p_1, \\ \mathit{equal}^{t,u} &= (Q^t = Q^u). \end{aligned}$$

There are four operations that can be applied to the queue at a time step. The *dequeue* operation removes the first pending element p_1, and

new elements are *enqueued* after p_n. If *deferring* occurs, then the element p_1 is removed and appended to the deferred elements. All defer operations can be undone by *flushing* the deferred elements back to the front of the pending elements list. These operations are defined by the following transformation functions. Below, $Q = \langle\langle d_1, \ldots, d_m\rangle, \langle p_1, \ldots, p_n\rangle\rangle$ is a queue and p is an arbitrary element.

$$\begin{aligned}
\mathit{dequeued}(Q) &:= \langle\langle d_1, \ldots, d_m\rangle, \langle p_2, \ldots, p_n\rangle\rangle,\\
\mathit{deferred}(Q) &:= \langle\langle d_1, \ldots, d_m, p_1\rangle, \langle p_2, \ldots, p_n\rangle\rangle,\\
\mathit{flushed}(Q) &:= \langle\langle\rangle, \langle d_1, \ldots, d_m, p_1, \ldots, p_n\rangle\rangle,\\
\mathit{enqueued}(Q, p) &:= \langle\langle d_1, \ldots, d_m\rangle, \langle p_1, \ldots, p_n, p\rangle\rangle.
\end{aligned}$$

The client-controlled Boolean variables deq^t, $defer^t$, $flush^t$, and enq^t determine whether a dequeue, defer, flush, or enqueue operation occurs at time step t, respectively. The variable $newelem^t$ of type ELEM determines the enqueued element, if any. When moving from time step t to $t+1$, the new contents of the queue is evaluated according to the following sequence of assignments.

$$\begin{aligned}
&Q \leftarrow Q^t\\
&\text{if } deq^t \text{ then } Q \leftarrow \mathit{dequeued}(Q)\\
&\text{if } defer^t \text{ then } Q \leftarrow \mathit{deferred}(Q)\\
&\text{if } flush^t \text{ then } Q \leftarrow \mathit{flushed}(Q)\\
&\text{if } enq^t \text{ then } Q \leftarrow \mathit{enqueued}(Q, newelem^t)\\
&Q^{t+1} \leftarrow Q
\end{aligned}$$

Thus, each of the four operations on the queue can occur once per time step, and if several operations occur, their order is fixed as listed above. Furthermore, the accessors $empty^t$, $pending^t$, $full^t$, and $firstelem^t$ are evaluated before applying the operations. The fixed order is chosen to follow a typical sequence of operations in a discrete execution step, in particular, firing a single transition of a UML state machine. In the beginning of such a step, the first pending message in an event queue is examined to check if it triggers a transition. Then, the message is possibly dequeued or deferred. If a transition is fired, the deferred messages are flushed because they potentially trigger a transition in the next step. Finally, the effect of the transition might send new messages that are enqueued in the appropriate queues.

A dequeue or defer operation is only possible if the queue has at least one pending element. An enqueue operation must only occur if the queue

Table 4.1. How a queue $Q^t = \langle\langle d_1, \ldots, d_m\rangle, \langle p_1, \ldots, p_n\rangle\rangle$ evolves to Q^{t+1} depending on the control variables. The symbols 0, 1, and $\times$ denote false, true, and don't care, respectively.

deq^t	$defer^t$	$flush^t$	enq^t	Q^{t+1}
0	0	0	0	$\langle\langle d_1, \ldots, d_m\rangle, \langle p_1, \ldots, p_n\rangle\rangle$
1	0	0	0	$\langle\langle d_1, \ldots, d_m\rangle, \langle p_2, \ldots, p_n\rangle\rangle$
0	1	0	0	$\langle\langle d_1, \ldots, d_m, p_1\rangle, \langle p_2, \ldots, p_n\rangle\rangle$
0	$\times$	1	0	$\langle\langle\rangle, \langle d_1, \ldots, d_m, p_1, \ldots, p_n\rangle\rangle$
1	0	1	0	$\langle\langle\rangle, \langle d_1, \ldots, d_m, p_2, \ldots, p_n\rangle\rangle$
0	0	0	1	$\langle\langle d_1, \ldots, d_m\rangle, \langle p_1, \ldots, p_n, newelem^t\rangle\rangle$
1	0	0	1	$\langle\langle d_1, \ldots, d_m\rangle, \langle p_2, \ldots, p_n, newelem^t\rangle\rangle$
0	1	0	1	$\langle\langle d_1, \ldots, d_m, p_1\rangle, \langle p_2, \ldots, p_n, newelem^t\rangle\rangle$
0	$\times$	1	1	$\langle\langle\rangle, \langle d_1, \ldots, d_m, p_1, \ldots, p_n, newelem^t\rangle\rangle$
1	0	1	1	$\langle\langle\rangle, \langle d_1, \ldots, d_m, p_2, \ldots, p_n, newelem^t\rangle\rangle$

has space—either a dequeue operation has freed a slot or the queue was not full to start with. Also, we do not allow both dequeuing and deferring an element at the same step. We assume that the client of the queue interface enforces these constraints by fulfilling the following invariants at every time step t.

$$deq^t \rightarrow pending^t,$$
$$defer^t \rightarrow pending^t,$$
$$\neg(deq^t \wedge defer^t),$$
$$enq^t \rightarrow (\neg full^t \vee deq^t).$$

To make the evolution of the queue more explicit, Table 4.1 lists all possible combinations of the control variables at a single time step, under the above restrictions, and the corresponding total effects on the queue.

Publication [**II**] describes a subset of this interface with no deferring support. The subset is obtained by regarding the control variables $defer^t$ and $flush^t$ as always false and the list of deferred elements $\langle d_1, \ldots, d_m\rangle$ as always empty. In this case, the accessor $pending^t$ is equivalent to $\neg empty^t$.

4.2 The Queue Encodings

In this section, we will go through the queue encoding approaches of Publications [**II**] and [**I**]. The *shifting* encoding represents a bounded queue as a finite vector of variables, where the ith variable holds the ith oldest element of the queue. Dequeuing an element then involves shifting the

remaining elements one step towards the beginning of the vector. The *cyclic* encoding is a variant that includes a *head* pointer to the oldest element. Upon dequeuing, instead of shifting the queue contents, the head pointer is incremented. In the cyclic encoding, the queue contents wrap around the end of the vector. The *linear* encoding also employs a head pointer, but instead of wrap-around semantics, the queued elements are placed in a sequence that extends to infinity. With the use of uninterpreted functions, the formula size of the linear encoding is independent of the capacity of the queue. This is also the only encoding that can accommodate unbounded queues. These three encodings are presented in Sects. 4.2.1 to 4.2.3.

In Sect. 4.2.4, we will discuss a technique that can be used on top of the shifting or cyclic encoding to reduce the encoding overhead in SMT-based bounded model checking, when the queue elements are composite objects that cannot be represented as a single term in the formula. The idea is not to directly store elements in the queue, but *tags*, which work like references to elements.

4.2.1 A Shifting Approach

The shifting queue encoding employs a straightforward representation of the sequence of queued elements $\langle\langle d_1, \ldots, d_m\rangle, \langle p_1, \ldots, p_n\rangle\rangle$ at each time step. The notation below is compatible with [**II**] but extended to accommodate the defer and flush operations.

For representing a queue with a bounded capacity Z at time step t, we introduce a sequence of variables $qc_0^t, \ldots, qc_{Z-1}^t$ of type ELEM. These variables hold the deferred elements $d_1, \ldots, d_m$ followed by the pending elements $p_1, \ldots, p_n$, as illustrated in Fig. 4.1(a). The timed integer variables $firstpos^t$ and $tail^t$ denote the zero-based index of the position just after the deferred elements and the pending elements, respectively. Thus, $0 \leq firstpos^t \leq tail^t \leq Z$ always holds.

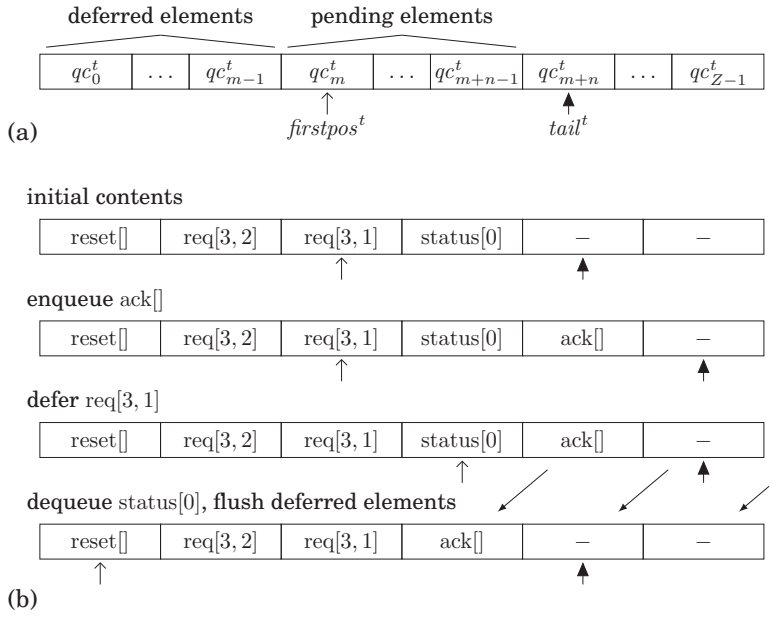


Figure 4.1. Representation of the queue contents in the shifting queue encoding, and an example evolution of a queue.

The accessors of the queue interface are then defined as follows.

$$empty^t := (tail^t = 0), \tag{4.1}$$

$$pending^t := (firstpos^t \neq tail^t), \tag{4.2}$$

$$full^t := (tail^t = Z), \tag{4.3}$$

$$firstelem^t := \begin{cases} firstpos^t = 0 & : qc_0^t \\ firstpos^t = 1 & : qc_1^t \\ \quad\vdots & \quad\vdots \\ firstpos^t = Z-2 & : qc_{Z-2}^t \\ \text{else} & : qc_{Z-1}^t, \end{cases} \tag{4.4}$$

$$equal^{t,u} := (firstpos^t = firstpos^u) \wedge (tail^t = tail^u) \wedge \bigwedge_{0 \leq s < Z} (s < tail^t) \rightarrow (qc_s^t = qc_s^u). \tag{4.5}$$

The notation on the right-hand side of (4.4) denotes a nesting of if-then-else constructs. For example, if $Z = 3$, then $firstelem^t$ is defined as the expression (if $firstpos^t$=0 then qc_0^t else (if $firstpos^t$=1 then qc_1^t else qc_2^t)).

Figure 4.1(b) illustrates the evolution of a queue under this encoding. When a new element (ack[] in the figure) is enqueued, it is placed at the current tail position, and the *tail* pointer is incremented. A defer operation is implemented as just incrementing *firstpos*. A dequeue operation

removes the element at $\mathit{firstpos}$ and shifts the subsequent elements one position lower (the diagonal arrows in the figure). A flush operation moves all deferred elements in front of the pending elements, which means resetting $\mathit{firstpos}$ to zero. The last transition in Fig. 4.1(b) involves dequeuing and flushing in the same step. In the general case, the transition from time step t to $t+1$ is encoded as follows.

$$qc_s^{t+1} := \begin{cases} \mathit{enq}^t \wedge \neg \mathit{deq}^t \wedge \mathit{tail}^t{=}s & : \mathit{newelem}^t \\ \mathit{enq}^t \wedge \mathit{deq}^t \wedge \mathit{tail}^t{=}s{+}1 & : \mathit{newelem}^t \\ \mathit{deq}^t \wedge \mathit{firstpos}^t{\leq}s & : qc_{s+1}^t \\ \mathsf{else} & : qc_s^t, \end{cases} \tag{4.6}$$

$$\mathit{firstpos}^{t+1} := \begin{cases} \mathit{flush}^t & : 0 \\ \mathit{defer}^t & : \mathit{firstpos}^t + 1 \\ \mathsf{else} & : \mathit{firstpos}^t, \end{cases} \tag{4.7}$$

$$\mathit{tail}^{t+1} := \begin{cases} \mathit{deq}^t \wedge \neg \mathit{enq}^t & : \mathit{tail}^t - 1 \\ \neg \mathit{deq}^t \wedge \mathit{enq}^t & : \mathit{tail}^t + 1 \\ \mathsf{else} & : \mathit{tail}^t. \end{cases} \tag{4.8}$$

Equation (4.6) above is instantiated for all indices $0 \leq s < Z$. In the boundary case $s = Z{-}1$, the term qc_{s+1}^t can be taken to have any constant value of the element type ELEM. In Fig. 4.1(b), the symbol "$-$" denotes this value.

Variants. The shifting encoding defined above uses integer variables for the pointers $\mathit{firstpos}^t$ and tail^t. As these integers have a bounded domain, they can also be encoded using bitvectors, thus enabling SAT and BDD-based methods.

As an alternative, we consider a *one-hot* encoding, in which tail^t is replaced by a sequence $\mathit{tail}_0^t, \ldots, \mathit{tail}_Z^t$ of Boolean variables, where each tail_s^t has the meaning $\mathit{tail}^t = s$. The variables $\mathit{firstpos}_0^t, \ldots, \mathit{firstpos}_Z^t$ are analogous. The motivation is to simplify the encoding so that a decision procedure for integers or bitvectors is no longer needed. As the pointers are mostly compared to constants, this change does not significantly increase the encoding size.

We obtain the version with one-hot pointer variables by making the following changes to the encoding. In the definitions of empty^t (4.1), full^t (4.3),

$\mathit{firstelem}^t$ (4.4), and qc_s^{t+1} (4.6), we replace all subformulas of the form $\mathit{firstpos}^t = j$ and $\mathit{tail}^t = j$ by $\mathit{firstpos}_j^t$ and tail_j^t, respectively. The definition (4.2) becomes

$$\mathit{pending}^t := \neg \bigvee_{0 \le s \le Z} (\mathit{firstpos}_s^t \wedge \mathit{tail}_s^t).$$

In (4.6), a special treatment is needed for the subformula $\mathit{firstpos}^t \le s$. In the case $s = 0$, this maps to $\mathit{firstpos}_0^t$. If $0 < s < Z$, we replace $\mathit{firstpos}^t \le s$ by $\big(\mathit{firstpos}_s^t \vee (\mathit{firstpos}^t \le s-1)\big)$ and apply this transformation recursively. Because the subformula $\mathit{firstpos}^t \le s-1$ is shared with the definition of qc_{s-1}^{t+1}, we can still express $qc_0^{t+1}, \ldots, qc_Z^{t+1}$ in size $\mathcal{O}(Z)$.

For the equality predicate, we employ a similar chain of definitions with a size linear in Z. For $0 \le s < Z$, we define an auxiliary predicate $\mathit{equal}_s^{t,u}$ as

$$\begin{aligned} \mathit{equal}_s^{t,u} := {} & (\mathit{tail}_s^t \wedge \mathit{tail}_s^u) \vee \\ & \big((\mathit{firstpos}_s^t \leftrightarrow \mathit{firstpos}_s^u) \wedge (qc_s^t = qc_s^u) \wedge \mathit{equal}_{s+1}^{t,u}\big) \end{aligned} \tag{4.9}$$

with the base case

$$\mathit{equal}_Z^{t,u} := (\mathit{tail}_Z^t \wedge \mathit{tail}_Z^u).$$

Then, $\mathit{equal}^{t,u} := \mathit{equal}_0^{t,u}$. Finally, the evolution of the variables $\mathit{firstpos}_s^t$ and tail_s^t for $0 \le s \le Z$ is determined as

$$\mathit{firstpos}_s^{t+1} := \begin{cases} \mathit{flush}^t & : (s = 0) \\ \mathit{defer}^t & : \mathit{firstpos}_{s-1}^t \\ \mathsf{else} & : \mathit{firstpos}_s^t, \end{cases}$$

$$\mathit{tail}_s^{t+1} := \begin{cases} \mathit{deq}^t \wedge \neg \mathit{enq}^t & : \mathit{tail}_{s+1}^t \\ \neg \mathit{deq}^t \wedge \mathit{enq}^t & : \mathit{tail}_{s-1}^t \\ \mathsf{else} & : \mathit{tail}_s^t. \end{cases}$$

The boundary cases $\mathit{firstpos}_{-1}^t$, tail_{-1}^t, and tail_{Z+1}^t are taken to have the value false, and the formula $(s = 0)$ is just the constant true or false.

Publication [**II**] presents the same one-hot encoding as above but omits deferring and $\mathit{firstpos}_s^t$. The pointer encoding in [**I**] is essentially a hybrid, where $\mathit{firstpos}^t$ is encoded as an integer (denoted by QPos in [**I**]), and tail^t is implicitly one-hot encoded. Namely, the unused slots in the vector representing the queue contents hold a special value $\mathbf{none} \in \textsc{Elem}$, and $\mathit{tail}^t{=}s$ is encoded as $(qc_s^t{=}\mathbf{none} \wedge qc_{s-1}^t{\neq}\mathbf{none})$.

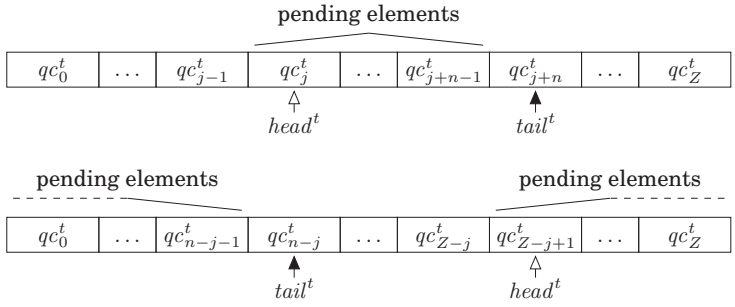


Figure 4.2. Representation of the queue contents in the cyclic queue encoding in the cases $head^t \leq tail^t$ and $head^t > tail^t$.

4.2.2 A Cyclic Approach

For an optimized implementation of a bounded first-in-first-out queue in an imperative programming language, a programmer would use a cyclic buffer, in which the current contents of the queue are represented as a window of successive elements that may wrap around at the end of the buffer. Our cyclic encoding employs the same idea. We only consider the case without deferring, thus the queue only contains pending elements.

Like in the shifting approach, the queue contents at time step t are represented by a finite vector of variables of type ELEM, with an integer variable $tail^t$ that points to the index where the next element will be enqueued. Another variable $head^t$ points to the oldest element in the queue. The pending elements are at indices from $head^t$ up to but not including $tail^t$, as shown in Fig. 4.2. In the case $head^t > tail^t$, the index wraps back to zero at the end of the array. The variables $head^t$ and $tail^t$ range from 0 to Z, inclusive. The index $tail^t$ in the vector never holds a queued element, and therefore we allocate a vector of length $Z + 1$ to accommodate at most Z pending elements. The queue is empty iff $head^t = tail^t$ and full iff $head^t \equiv tail^t + 1 \pmod{Z+1}$.

A dequeue operation, instead of shifting elements, increments the $head$ variable modulo $Z + 1$. An enqueue operation increments $tail$ modulo $Z + 1$. When moving from step t to $t + 1$, each variable qc_s^t either retains its old value or is replaced by the enqueued element $newelem^t$. As opposed to the shifting encoding (4.6), the definition of qc_s^{t+1} no longer depends on qc_{s+1}^t. The rationale for the cyclic approach is that it reduces the coupling between adjacent slots of the array. The encoding details can be found in [**II**].

To extend the cyclic encoding to support deferring, one might introduce a

third pointer variable $firstpos^t$ that marks the boundary between deferred and pending elements like in Fig. 4.1(a). However, a dequeue operation would then need to shift the *deferred* elements one position forward, in addition to incrementing $head$. Such mechanics would bring the encoding close to the shifting approach.

Variants. Analogously to the shifting approach, the $head^t$ and $tail^t$ pointers of the cyclic encoding can also be encoded using bitvectors or one-hot Boolean variables instead of integers.

Another variant, eligible in SMT-based BMC, is to discard the explicit variables $qc_0^t, \ldots, qc_Z^t$ that hold the queued elements and use uninterpreted functions (uifs) instead. For each time step t, we define an uninterpreted function $\mathsf{qc}^t : \textsc{Int} \to \textsc{Elem}$. In the encoding, we replace each occurrence of qc_s^t by $\mathsf{qc}^t(s)$, i.e. the uninterpreted function qc^t applied to an integer constant. The benefit is that the first pending element can be queried compactly as $firstelem^t := \mathsf{qc}^t(head^t)$ instead of an explicit case split over the $Z + 1$ possible values of $head^t$. The drawback is increased complexity due to the introduction of function symbols instead of just variables.

An even higher-level representation relies on satisfiability modulo the theory of arrays introduced in Sect. 2.1. An array variable $qc^t : \textsc{Int} \to \textsc{Elem}$ represents the queue contents at time step t. The contents at the next step is then $qc^{t+1} := \mathsf{if}\ enq^t\ \mathsf{then}\ write(qc^t, head^t, newelem^t)\ \mathsf{else}\ qc^t$. This constant-size expression eliminates the need for separately updating each array slot. The overall size of the encoding (disregarding the equality formulas $equal^{t,u}$) is constant per time step and does not depend on the capacity Z. The compactness relies on the fact that at most one array element changes its value between successive time steps. Therefore, the shifting approach would not lend itself to such a succinct array encoding.

The cyclic encoding with one-hot pointers and with uif- or array-based contents representation are explicitly defined in [**II**].

4.2.3 A Linear Approach

Our third, SMT-based approach is called the linear encoding. The idea is to take the cyclic approach and let the length of the buffer grow indefinitely. At the limit, the $tail$ pointer never wraps around, and every

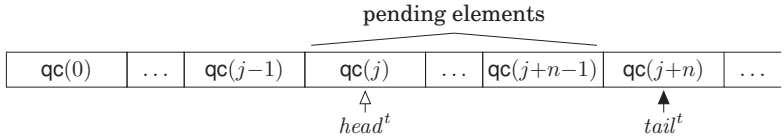


Figure 4.3. Representation of the queue contents in the linear queue encoding.

enqueued element is stored in a fresh slot. An essential observation is that during the entire evolution of the queue, each slot holds at most one value of type ELEM. To model this case, we can use a single, *time-independent* uninterpreted function qc : INT $\rightarrow$ ELEM to represent the contents. The only time-dependent variables are $head^t$ and $tail^t$. This is depicted in Fig. 4.3.

We get a compact encoding, as there are no frame conditions for tying together the queued elements between successive time steps. The entire dynamics is captured by the following constraints for each time step. See [**II**] for the complete encoding.

$$head^{t+1} := \text{if } deq^t \text{ then } head^t + 1 \text{ else } head^t, \tag{4.10}$$

$$tail^{t+1} := \text{if } enq^t \text{ then } tail^t + 1 \text{ else } tail^t, \tag{4.11}$$

$$enq^t \rightarrow \big(\mathsf{qc}(tail^t) = newelem^t\big). \tag{4.12}$$

Unless we explicitly limit the number of pending elements $tail^t - head^t$ to the range $0, \ldots, Z$, this encoding automatically models an unbounded queue. Nevertheless, in a finite number of time steps with at most one enqueue operation per step, the queue cannot grow indefinitely.

We do not handle deferring in the linear encoding. Allowing the flush operation would lead to a non-linear dequeuing order of elements. With the time-independent representation of the queue contents, there is no obvious way to implement this.

4.2.4 Compressing Tuple Elements with Tags

In modeling languages such as the input languages of the Spin [51] and DiVinE [9] model checkers and our UML subset (Sect. 3), communication may involve messages that carry data parameters. When storing such messages in a queue, the type of the elements is composite instead of scalar. In a software implementation, communicating such messages between components could be done by passing references to objects instead of contents, for the sake of efficiency. We adapt a similar optimization to SMT-based BMC. In our encoding, uninterpreted function application

plays the role of object dereferencing.

For simplicity, let us assume that the element type is a tuple of scalar types $T_1, \ldots, T_A$. For example, messages that consist of a signal identifier and at most two integer parameters can be represented as tuples of the form $\langle \mathit{signal}, \mathit{param1}, \mathit{param2} \rangle \in$ TUPLE(SIGNAL, INT, INT). Encoding such tuples introduces new variables and expressions. In the shifting encoding, the variable qc_s^t that holds the sth oldest queued element at time step t is expanded to A separate variables of types $T_1, \ldots, T_A$, respectively. How these variables evolve in time (the definition (4.6) of qc_s^{t+1}) is duplicated A times as well. The logic that controls the evolution of the first tuple element is the same as the logic for the $A - 1$ remaining elements, and we would like to factor out the redundant parts of the encoding. The duplication also applies to the cyclic encoding, and to the array- and uif-based representations of queue contents.

Naturally, the duplication can be avoided if the back-end solver directly supports terms whose type is a tuple. For example, the Z3 and CVC3 solvers have native tuple types, but the SMT-LIB language [85] does not. For the cases where tuple-valued variables are not available, publication **[II]** proposes an alternative that avoids storing tuples in the queue. The idea is to instead store scalar elements (say, integers) that we call *tags*. Tags act as references to tuples. For dereferencing, we define time-independent uninterpreted functions $\mathit{decode}_1, \ldots, \mathit{decode}_A$. Then, we have $\mathit{decode}_i(r) = x$ if tag r refers to a tuple whose ith element is x. Enqueuing a tuple value transforms to enqueuing a tag whose concrete value is unspecified, while ensuring that the enqueued tag refers to the correct tuple. For example, to enqueue the message $\langle \mathrm{status}, 0 \rangle$ at step t, the client places the constraint

$$\mathit{enq}^t \wedge \big(\mathit{decode}_1(\mathit{newelem}^t) = \mathrm{status}\big) \wedge \big(\mathit{decode}_2(\mathit{newelem}^t) = 0\big).$$

Similarly, the predicate "the first pending element at step t is $\langle \mathrm{status}, 0 \rangle$" is encoded as

$$\mathit{pending}^t \wedge \big(\mathit{decode}_1(\mathit{firstelem}^t) = \mathrm{status}\big) \wedge \big(\mathit{decode}_2(\mathit{firstelem}^t) = 0\big).$$

With this technique, the encoding of how the queue evolves in time becomes independent of the arity A of the element type. The dependence on A cannot be completely eliminated because the tuples need to be deconstructed at the interface, as shown above. The tag-based compression can be applied to all encodings of this section except the linear encod-

ing of Sect. 4.2.3, which is immune to the problem because of the time-independent representation of queued elements.

The use of tags as object references might find more general use in SMT encodings of structured data. The tradeoff is that it is cheaper to move or copy tags than entire structures, but accessing the data in the structure involves an extra level of indirection in the form of an uninterpreted function application. It is worth noting that our tags do not fulfill the property of extensionality. That is, two different tags may refer to the same tuple, and the equality of tuples cannot be resolved by comparing tags alone. For this reason, the use of tags does not compress the $equal^{t,u}$ predicates.

4.3 Related Work

Our compact cyclic array-based representation of Sect. 4.2.2 can be extended to fully simulate a first-in-first-out queue datatype, without restricting to a non-branching evolution like in our BMC context. Such an approach is used as an SMT solving benchmark in a recent work by Armando et al. [5]. A reduction of queues to arrays and head/tail pointers is defined in [5] both with cyclic (like Sect. 4.2.2) and unbounded semantics (like Sect. 4.2.3).

In the thesis of Bjørner [12], a decision procedure for a theory of queues with an extensive set of operations is developed. See also the related work section in [**II**].

4.4 Discussion

To enable symbolic model checking of systems that include queues e.g. in the form of buffered communication, publication [**II**] presents three alternative approaches to encode queues especially in the context of SMT-based bounded model checking. The encodings are summarized in a feature matrix in Table 4.2. The three approaches are further subdivided based on whether the queued elements are represented explicitly using separate variables, or using arrays or uninterpreted functions (uifs). Furthermore, the variables that act as pointers to elements can be either (bounded) integers or one-hot encoded Booleans.

All encodings can model basic first-in-first-out queues. In addition, the

shifting approach is augmented with support for the defer operation of UML state machines. The encodings are restricted to bounded queues of a fixed capacity Z, except for the linear approach, which also supports unbounded queues.

The encodings require different theories to be supported by the back-end SMT solver, as shown in Table 4.2. The encodings with one-hot pointers do not rely on any theories and could thus be used directly in SAT- and BDD-based verification. Integer pointers require support for the theory of integer offsets [5, 74], or more generally, integer linear arithmetic. The encodings with uif or array contents representation require support for the theory combination of integer offsets with uifs or arrays [94], respectively.

In a BMC instance, the size of the shifting or cyclic encoding with explicit or uif-based contents representation is linear in the capacity Z times the bound k. However, if the element type is a tuple of A scalars, the size expands to $\mathcal{O}(kZA)$. This can be compressed to $\mathcal{O}(k(Z + A))$ by using the tag technique of Sect. 4.2.4. The size of the cyclic encoding with array-based contents, as well as the linear encoding, is independent of Z. With tuple elements, the size of these encodings is $\mathcal{O}(kA)$, regardless of whether tags are used. The equality predicates are not included in the sizes above, as they are not needed in checking invariant properties.

In the experiments of [**II**], the approaches are compared to each other on artificial BMC tests as well as benchmarks translated from simple UML models. Generally, the cyclic encoding with explicit contents representation and one-hot encoded $head$ and $tail$ pointers performs best. Compared to the shifting approach, the evolution of individual queue slots is expressed with slightly simpler formulas, and this seems to outweigh the

Table 4.2. Queue encoding variants and the supported features.

approach	shifting		cyclic				linear
contents	explicit		explicit		uif	array	uif
pointers	int	one-hot	int	one-hot	int	int	int
features							
deq, enq	•	•	•	•	•	•	•
defer, flush	•	•					
unbounded							•
encoding							
theories	int	–	int	–	int + uif	int + array	int + uif
size	$\mathcal{O}(kZ)$	$\mathcal{O}(kZ)$	$\mathcal{O}(kZ)$	$\mathcal{O}(kZ)$	$\mathcal{O}(kZ)$	$\mathcal{O}(k)$	$\mathcal{O}(k)$
tags	•	•	•	•	•	•	

potential confusion caused by the rotational symmetry of the vector that represents the queue contents. The linear encoding, although compact, resulted in the longest solving times. The use of tags seems to consistently give at least some speedup compared to encodings that duplicate the logic for each tuple element. Significant improvements were observed with models that involved tuples with a large arity.

All in all, a precise symbolic encoding of the semantics of a queue results in relatively heavyweight formulas. As with other software features, a viable approach might be conservative abstraction of the semantics. For example, one conceivable approximation would be to abstract from the ordering of elements and to treat the queue contents as a multiset instead of an ordered sequence of elements.

5. Symbolic Partial Order Methods

In a distributed system, local events in different components occur concurrently, and it is generally not meaningful or even possible to investigate the order of these occurrences. Our model of systems, based on global states and interleaving executions, has the disadvantage that it forces a total order on unrelated as well as related occurrences.

The theory of Mazurkiewicz traces [66] offers a view of executions that only fixes a *partial order* of occurrences. An interleaving execution is represented as a finite string of symbols. The symbols correspond to the possible actions of the system, and each pair of symbols is either *dependent* or *independent*. The exact definition of independence may vary, but a minimum requirement is that if two actions a and b are independent, then $a \neq b$, executing a in any reachable state does not change the enabledness of b or vice versa, and in every reachable state where both actions are enabled, both execution orders ab and ba lead to the same state. Then a string $wabw'$, where w and w' are arbitrary strings, is equivalent to the string $wbaw'$. More generally, two strings are equivalent iff one can be obtained from the other by a number of transpositions of subsequent independent symbols. An equivalence class of strings is called a *trace*.

When checking safety properties, examining two executions in the same trace is redundant because they are guaranteed to end up in the same state. In explicit-state model checking, this idea is employed in the form of partial order reduction [44]. Transitions in the state space are pruned from the search while guaranteeing that at least one representative execution for each Mazurkiewicz trace remains. This reduction often leads to tremendous savings in run time, due to the inherent abundance of independent actions in concurrent systems.

A major focus of this thesis is to apply partial order methods to accelerate bounded model checking of concurrent systems. However, the ap-

proach is quite different from the reduction used in explicit-state model checking. In BMC, a bottleneck is that when the bound increases, the time to check the satisfiability of the unrolled formula grows rapidly. Increasing the bound by one involves encoding a new copy of every action into the formula, even though only one of the actions is executed in any concrete execution under the interleaving semantics. In [**III**] and [**IV**], we address the bottleneck of BMC by employing so-called *alternative execution semantics*, while [**V**] introduces a different partial order technique called *bounded event tracing*. These approaches are briefly introduced below. Although we talk about alternative semantics, our principle is to conform to the interleaving semantics in terms of the reachability of states.

- The idea of *step semantics* is to allow each execution step to contain not just one action, but any set of independent actions. In essence, step semantics adds shortcut edges to the state space, allowing a given state to be reached in fewer steps. Therefore, we can cover a larger portion of the state space at each BMC bound than with the interleaving semantics, without mentionable expansion in the formula size. This makes it significantly faster to find counterexample executions to safety properties in practice. The variants we consider, called the *parallel* $\exists$*-step semantics* and the *serial* $\exists$*-step semantics*, are discussed in Sect. 5.1. These two semantics have been presented earlier in the context of SAT-based planning [86] and SAT-based model checking of 1-safe Petri nets [77], respectively. The contribution of [**III**] is to extend parallel $\exists$-steps to object-based systems with data variables and message queues. In [**IV**], parallel and serial $\exists$-steps are treated in a unified abstract framework that can accommodate many modeling languages of concurrent systems.

- While the shortcut edges of step semantics add redundant paths to the state space, the essence of *process semantics* is to only allow execution paths in a certain normal form. Process semantics has been employed before in BMC for Petri net [48] and labeled transition system [58] formalisms. Section 5.2 discusses the *serial process semantics*, which is a new contribution that combines the idea of process semantics and serial $\exists$-steps. We will show that the serial process semantics is optimal in the sense that it allows exactly one execution in each Mazurkiewicz trace.

- *Bounded event tracing* (Sect. 5.3) is a novel SMT-based technique for

checking bounded safety properties of concurrent systems. It differs from bounded model checking as there is no unrolling of a transition formula that encodes the actions of the system. Instead, we take a fixed number of potential events, which correspond to unrolled copies of actions, and construct a formula that characterizes all executions where each potential event occurs at most once. The order of the occurrences is unspecified, but constraints are added to the formula to make sure that a cause always precedes its effect. Therefore, bounded event tracing is inherently a partial order method.

We will go through these contributions in the following sections, with related work in Sect. 5.4 and concluding discussion in Sect. 5.5.

5.1 Step Semantics

In the interleaving semantics, each execution step is a *unit step*: exactly one action is executed. In BMC, the transition formula is unrolled k times to cover the part of the state space reachable from the initial state within k unit steps. Recall that an interleaving transition formula (2.5) contains an encoding of each action that can occur. Thus, the unrolled formula contains a copy of every action encoded at every step. In a sense, this is wasteful, since only one action per time step is scheduled for execution. For single-threaded systems, we could employ analysis based on the control flow graph to statically reduce the possible scheduling choices at each step [39, 27]. Unfortunately, such pruning does not scale to systems with concurrency. Because of the nondeterministic interleaving of actions from different components, we cannot in practice say that at a certain global time step, we only need to consider a small set of actions local to a component.

Instead, the approach presented in [**III**] and [**IV**] is to allow several actions to be executed in each step. In particular, the idea of *step semantics* is to preserve all unit steps and also add shortcut edges to the state space such that each shortcut edge corresponds to a sequence of unit steps. The transition formula is rewritten so that a given BMC bound k covers all states reachable within k unit steps, and usually a significantly larger part of the reachable state space as well. This can be very beneficial, as the satisfiability solver run time on BMC instances is in practice super-

linear or even exponential in the bound. Naturally, we still do not want to outweigh the benefits by adding too much complexity to the formula. By exploiting the inherent independence of actions, a modest modification to the interleaving transition formula suffices to implement a step semantics that, in particular, allows all steps consisting of a set of enabled, pairwise independent actions.

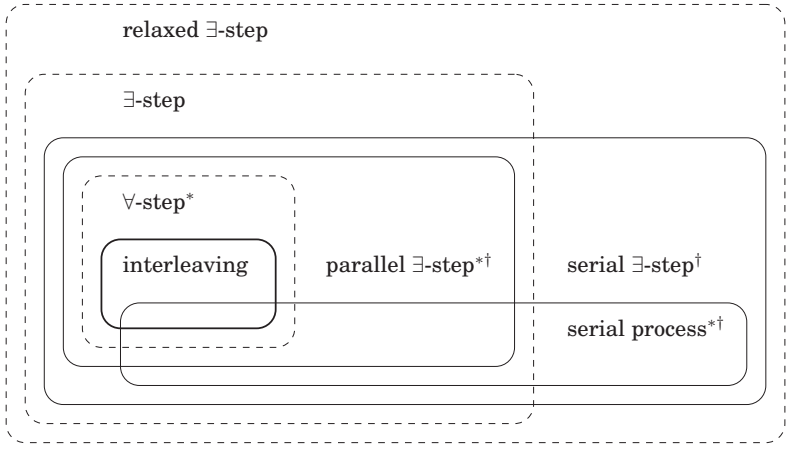
In the following, we will discuss step semantics in general (Sect. 5.1.1), a way to represent actions (Sect. 5.1.2), the serial and parallel ∃-step implementations (Sects. 5.1.3–5.1.4), experiments (Sect. 5.1.5), and approximation of independence (Sect. 5.1.6).

5.1.1 Semantic Definitions

Let us present three alternative execution semantics: the ∀-*step*, ∃-*step*, and *relaxed* ∃-*step semantics*. The semantics actually employed for model checking in this work do not coincide with any of these, but are closely related. As a baseline, we take the interleaving state space of the system. Under the alternative execution semantics, the state space contains the same set of states but a different set of transitions. There is a transition from state s to state s' labeled with a nonempty set of actions ex if the actions in ex are executable in some order starting from s and reaching s', with the following restrictions specific to each semantics.

- Interleaving semantics: ex contains exactly one action enabled in s.
- ∀-step semantics: the actions in ex are enabled in s and pairwise independent.
- ∃-step semantics: the actions in ex are enabled in s.
- Relaxed ∃-step semantics: no restrictions.

The semantics above are listed in order from the most restrictive to the most liberal. The ∀-step semantics is included as it corresponds to the classical step semantics from Petri net theory [53]. However, for BMC, we employ the even less restrictive ∃-step and relaxed ∃-step semantics. Under the ∃-step semantics, there *exists* an ordering of the actions in a step that reaches the target state, while ∀-step semantics requires that



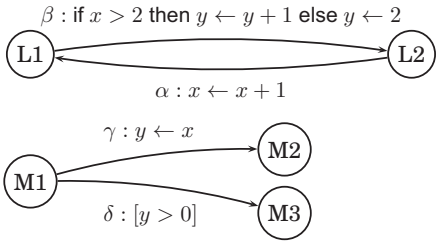
* Subject to approximation in the definition of independence.
† Subject to the chosen total order of actions.

Figure 5.1. Interleaving and alternative execution semantics in the set of all possible execution paths. The solid lines denote the semantics implemented for BMC in this work.

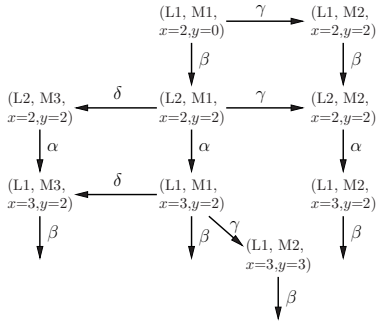
all orderings lead to the same state. In general, every interleaving execution is a ∀-step execution, every ∀-step execution is an ∃-step execution, and every ∃-step execution is a relaxed ∃-step execution. In the other direction, every relaxed ∃-step execution can be turned into an interleaving execution by expanding the relaxed ∃-steps into sequences of unit steps. This means that the set of reachable states from given initial states coincides for all four semantics. Figure 5.1 illustrates the relationship of these four semantics, as well as other semantics that will be explained in later sections.

Example 1. Consider the system in Fig. 5.2(a). The system has two components with control locations $\{\mathrm{L1}, \mathrm{L2}\}$ and $\{\mathrm{M1}, \mathrm{M2}, \mathrm{M3}\}$, and four actions $\{\alpha, \beta, \gamma, \delta\}$. The state variables are pc_L and pc_M, which hold the current control location of the components, and integers x and y. Figure 5.2(b) shows a part of the interleaving state space of the system. In Fig. 5.2(c), the dashed arrows together with the solid arrows denote the transitions under the ∃-step semantics, and all arrows together correspond to the relaxed ∃-step semantics. If we consider action α to be independent of δ, then the ∀-step semantics corresponds to the solid arrows plus the transition from state $(\mathrm{L2}, \mathrm{M1}, x{=}2, y{=}2)$ to $(\mathrm{L1}, \mathrm{M3}, x{=}3, y{=}2)$.

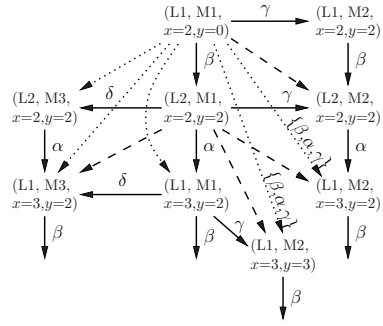
As a motivation for the step semantics approach, observe that starting from the initial state $(\mathrm{L1}, \mathrm{M1}, x{=}2, y{=}0)$, it takes 3 unit steps to reach the



(a) control flow graphs of the two components of the system



(b) interleaving state space



(c) relaxed ∃-step state space

Figure 5.2. An example system and parts of its interleaving and relaxed ∃-step state spaces.

state $(\mathrm{L1}, \mathrm{M2}, x{=}3, y{=}2)$. This can be compressed to 2 ∀-steps or 2 ∃-steps, or to 1 relaxed ∃-step. The latter compression is possible because α can first become enabled in a relaxed ∃-step and then be executed in the same step.

It does not seem practical to realize the full relaxed ∃-step semantics in symbolic model checking. Therefore, our goal is to define a step semantics transition formula that includes a large number of the transitions in the relaxed ∃-step state space but is not much larger in size than the interleaving transition formula. Two such realizations are presented in the following sections.

5.1.2 Representing Actions

Like in [**IV**], we assume that the states of the system are given as vectors that give values to a finite set of typed *state variables* V, and the behavior is defined by a finite set of actions $\mathsf{act}^1, \ldots, \mathsf{act}^n$. We take a fine-grained view where the independence of two actions is not statically fixed, but may be sensitive to the current state or possible nondeterministic choices. Therefore, we break down the behavior of each action into *ground ac-*

tions, in which all data (values from state variables and nondeterministic choices as well as values computed by the action) is fixed to constants.

Definition 1. A ground action g is a triple $\langle \mathsf{act}, R, W \rangle$, where

- act identifies the action,
- R is a set of *guards* of the form $[v = C]$, where $v \in V$ and C is a constant in the domain of v, and
- W is a set of *assignments* of the form $v \leftarrow C$ such that W contains at most one assignment to each state variable.

A ground action is *enabled* in a state iff all its guards are satisfied in the state. An enabled ground action is *executed* atomically by applying its assignments in parallel. We associate each action act to a (potentially infinite) set of ground actions, denoted by $gnd(\mathsf{act})$.

Example 2. Consider the action γ in Fig. 5.2(a), which executes the statement $y \leftarrow x$. The action reads state variables pc_M and x, and assigns pc_M and y. We can represent γ as the set of ground actions

$$gnd(\gamma) = \left\{ \langle \gamma, \{[pc_M = \mathbf{M1}], [x = C]\}, \{pc_M \leftarrow \mathbf{M2}, y \leftarrow C\} \rangle \;\middle|\; C \in \textsc{Int} \right\}.$$

We will use ground actions as a conceptual tool to make explicit how actions depend on state variables, without committing to any particular modeling language. Publication [**IV**] uses a more practicable representation of actions as sets of expressions that encode how the actions read and write state variables. These expressions are directly used as building blocks of transition formulas. The two manners of presentation are compatible, and there is a straightforward mapping from the encoding expressions of [**IV**] to sets of ground actions. The reason for choosing a higher-level presentation for this section is to make it easier to reason about steps and independence.

5.1.3 Serial ∃-Step Semantics

In practice, the model checking implementations of this work do not allow all relaxed ∃-steps of the semantic definitions above. To keep the transition formula size small, we only consider relaxed ∃-steps in which the order of execution respects a predefined total order $\mathsf{act}^1 \prec \cdots \prec \mathsf{act}^n$ of

actions. We call such steps *serial ∃-steps*, and [**IV**] describes a transition formula that allows exactly the transitions corresponding to these steps.

We represent a step as a string $(g^1 \dots g^n)$, where each g^i is a ground action in $gnd(\mathsf{act}^i)$ or, if act^i is not executed in the step, a special *skip* ground action. The skip ground action, denoted by -, is just a placeholder that has no guards and no assignments and is thus always enabled. The step is executed by executing the ground actions in the order $g^1, \dots, g^n$ corresponding to the order $\prec$. A step $(g^1 \dots g^n)$ or a sequence

$$(g^1 \dots g^n)(g^{n+1} \dots g^{2n}) \cdots (g^{(k-1)n+1} \dots g^{kn})$$

of k steps is *enabled* in a state s iff g^1 is enabled in s, and every further ground action g^i in the sequence is enabled after executing $g^1, \dots, g^{i-1}$ starting from s.

The *empty step* $(\text{-} \dots \text{-})$ consists of only skip ground actions, and a *unit step* contains exactly one non-skip ground action. Thus, an execution under the interleaving semantics is fully identified by an initial state and a sequence of unit steps.

Definition 2. A step $(g^1 \dots g^n)$ is a *serial ∃-step* iff it is not the empty step. A *serial ∃-step execution* consists of an initial state s_0 and a finite sequence of serial ∃-steps such that the sequence is enabled in s_0.

Example 3. Consider the system of Fig. 5.2(a) with the order $\beta \prec \gamma \prec \alpha \prec \delta$ of actions. In the initial state $(\mathrm{L1}, \mathrm{M1}, x{=}2, y{=}0)$, the step $(g\text{-}\text{-}h)$ is enabled, where $g = \langle \beta, \{[pc_L = \mathrm{L1}], [x = 2]\}, \{pc_L \leftarrow \mathrm{L2}, y \leftarrow 2\}\rangle \in gnd(\beta)$ and $h = \langle \delta, \{[pc_M = \mathrm{M1}], [y = 2]\}, \{pc_M \leftarrow \mathrm{M3}\}\rangle \in gnd(\delta)$. The step can be expanded to the sequence $(g\text{-}\text{-}\text{-})(\text{-}\text{-}\text{-}h)$ of unit steps.

Regarding the semantic definitions of Sect. 5.1.1, every interleaving execution is a serial ∃-step execution, and every serial ∃-step execution is a relaxed ∃-step execution, only with a fixed ordering of actions. These relationships are illustrated in Fig. 5.1. Notice that the serial ∃-steps are neither a subset nor a superset of the (non-relaxed) ∃-steps, which allow any execution order of actions, but require every action to be enabled in the beginning of the step. In Example 3 above, the action δ is not enabled in the initial state, but is still executable as part of a serial ∃-step from the initial state. This capability makes serial ∃-steps a powerful semantics, allowing system components to perform long chains of actions in a single step.

Publication [**IV**] presents a concrete definition of a serial ∃-step transition formula based on a set of expressions that describe the individual

actions. Under the serial $\exists$-step semantics, a transition from state $s = s^0$ to $s' = s^n$ corresponds to a sequence $s^0 \xrightarrow{g^1} s^1 \xrightarrow{g^2} s^2 \to \cdots \to s^{n-1} \xrightarrow{g^n} s^n$, where each g^i is a ground action in $gnd(\mathsf{act}^i) \cup \{\text{-}\}$. The transition formula is based on encoding the n sub-steps such that in the ith sub-step, either an instance of act^i is executed or nothing happens. The intermediate states $s^1, \ldots, s^{n-1}$ are explicitly encoded. Having to represent these intermediate states does not, however, bloat the formula size in the experiments of [**IV**]. Because actions usually involve assignments to only a few state variables, the common case is that most state variables are statically known to have the same value in s^i as in s^{i+1}, and thus the representation can be shared in the transition formula.

5.1.4 Parallel $\exists$-Step Semantics

Another semantics, implemented for symbolic model checking in [**III**] and in [**IV**], is based on parallel execution of a set of (partially) independent actions. We follow the principle that two actions are dependent iff they access the same resource and at least one of the accesses is a write. To allow two actions to be dependent in some states and independent in others, we define the independence relation between ground actions instead of actions.

Definition 3. A ground action g *contradicts* another ground action h iff g and h assign different values to some state variable. We say that g *affects* h iff some state variable occurs both in an assignment in g and in a guard in h. Two ground actions are *independent* iff neither affects the other, their sets of assigned variables are disjoint, and they do not belong to the same action. Two ground actions are *dependent* iff they are not independent. In particular, the skip ground action - is independent of any other ground action.

Observe that if g and h are enabled in a state and do not contradict each other, then we can take the union of the assignments in g and h and apply the assignments in parallel. If, in addition, g does not affect h, then the result of the parallel assignment is the same as executing first g and then h. This is the basis for the parallel $\exists$-step semantics. Like in Sect. 5.1.3, we fix a total order $\mathsf{act}^1 \prec \cdots \prec \mathsf{act}^n$ of actions and consider steps of the form $(g^1 \ldots g^n)$.

Definition 4. A step $(g^1 \ldots g^n)$ where each $g^i \in gnd(\mathsf{act}^i) \cup \{\text{-}\}$ is a *parallel* $\exists$*-step* iff it is not the empty step and for all $1 \leq i < j \leq n$, the ground

action g^i neither contradicts nor affects g^j. A *parallel ∃-step execution* consists of an initial state s_0 and a finite sequence of parallel ∃-steps that is enabled in s_0.

Example 4. Let us choose the order $\beta \prec \gamma \prec \alpha \prec \delta$ for the actions of the system of Fig. 5.2(a). In the state $(\mathrm{L2}, \mathrm{M1}, x{=}2, y{=}2)$, the ground actions $g = \langle \gamma, \{[pc_M = \mathrm{M1}], [x = 2]\}, \{pc_M \leftarrow \mathrm{M2}, y \leftarrow 2\}\rangle \in gnd(\gamma)$ and $h = \langle \alpha, \{[pc_L = \mathrm{L2}], [x = 2]\}, \{pc_L \leftarrow \mathrm{L1}, x \leftarrow 3\}\rangle \in gnd(\alpha)$ are enabled. As g does not assign any state variable that occurs in a guard in h, g does not affect h. The ground actions do not contradict each other, as they assign disjoint state variables. Therefore, the step $(\text{-}gh\text{-})$ is a parallel ∃-step. Like all parallel ∃-steps, it is also a serial ∃-step.

Generally, a parallel ∃-step $(g^1 \dots g^n)$ is enabled in a state s if and only if $g^1, \dots, g^n$ are all enabled in s. This means that parallel ∃-steps are ∃-steps in terms of the semantic definition of Sect. 5.1.1. Also, recall that under the ∀-step semantics, a step consists of a set of pairwise independent actions. As long as we commit to the above definition of independence, two independent ground actions never affect or contradict each other, hence every ∀-step execution is also a parallel ∃-step execution. These relations are depicted in Fig. 5.1.

The parallel ∃-step transition formula defined in [**IV**] builds on the interleaving transition formula with two modifications. First, the constraint that a step contains at most one action is removed, allowing several enabled ground actions to be executed in the same step, as long as their assignments to state variables are not contradictory (cf. removing the $one\text{-}hot$ constraint in (2.5)). Second, the requirement that no ground action is affected by an earlier ground action in the same step is realized by adding constraints of the following form: if an action act^j reads a state variable v, then there is no action $\mathsf{act}^i \prec \mathsf{act}^j$ that assigns v in the same step. The primary motivation of modifying the interleaving transition formula is to allow all ∀-steps, that is, to allow independent system components to execute actions in parallel. However, instead of the ∀-step semantics, we employ the parallel ∃-step semantics because it is less restrictive and has a compact encoding as a transition formula without the need to list all pairs of potentially dependent actions.

In a parallel ∃-step, each state variable can be changed at most once, and in particular, each system component can make just one move to a new control location. In this sense, the parallel ∃-step semantics is less ambitious than the serial ∃-step semantics. However, as the parallel ∃-

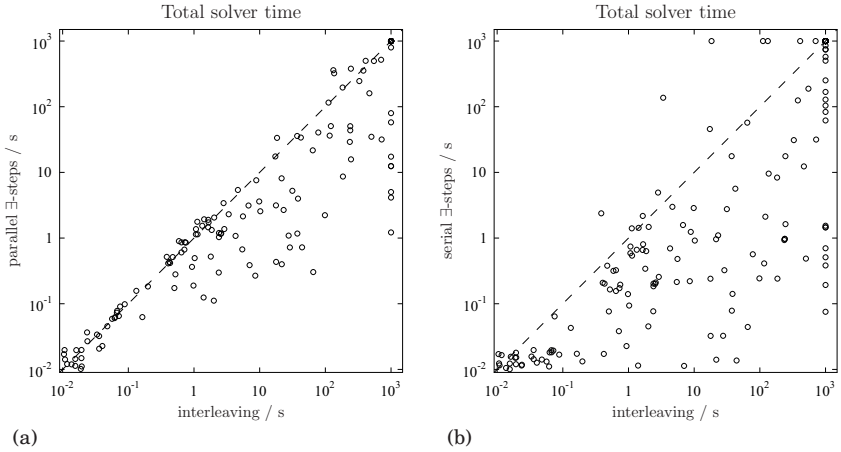


Figure 5.3. BMC efficiency with interleaving vs. step semantics transition formulas. Comparing cumulative BMC run times to find counterexample executions to invariant properties of BEEM benchmark instances.

step transition formula is based on executing the actions in parallel from the starting state, no intermediate states are encoded in the transition formula, and this simplicity with respect to the serial ∃-step transition formula might pay off in the satisfiability solving phase.

5.1.5 Experiments with Step Semantics

To evaluate the effect of step semantics, [**IV**] experimentally compares BMC with the parallel and serial ∃-step semantics to BMC with the interleaving semantics and to explicit-state model checking. This is done by taking the extensive model checking benchmark set BEEM [80] and automatically translating the models and properties to our abstract system formalism consisting of state variables and actions. The systems in BEEM are expressed as concurrent state machines that communicate using shared variables and rendezvous synchronization. The data types are 8- and 16-bit integers and fixed-length arrays. The benchmarks are checked for invariant properties, which are also taken directly from the BEEM distribution. As we employ BMC as an incomplete method, we get no definite results for benchmarks that satisfy the property.

Below are the main findings from these experiments.

- Switching from the interleaving to the *parallel* ∃-step semantics is beneficial for BMC performance. This is illustrated in Fig. 5.3(a), where each marker denotes a BEEM instance for which BMC finds a counterexam-

ple within a 1000 second time limit. The axes denote cumulative solver runtime in seconds starting from bound 0 until the bound with which the counterexample is found. The speed-up from the parallel $\exists$-step semantics is due to reduction in the required bound. However, in many instances the counterexample bound is not reduced, thus the shortest parallel $\exists$-step counterexample execution is also an interleaving execution. This explains the mass on the diagonal of Fig. 5.3(a). The experiments with the parallel $\exists$-step semantics in [**III**] are in line with these results.

- The *serial* $\exists$-step semantics generally far outperforms both the interleaving (see Fig. 5.3(b)) and the parallel $\exists$-step semantics. All but the most trivial (2 unit steps or less) interleaving counterexample executions are compressed to a significantly smaller number of serial $\exists$-steps.

- Switching from the interleaving to the parallel or serial $\exists$-step semantics does not significantly affect the size of the transition formula. On average, there is no more than $13\ \%$ increase in size.

- Even with the fastest semantics (serial $\exists$-steps), our BMC approach is generally slower on BEEM benchmarks than the explicit-state model checker DiVinE [9]. This is in part explained by the benchmarks having been written in the native input language of DiVinE—in fact, BEEM stands for "Benchmarks for *Explicit* Model checkers". There are, however, cases where BMC is orders of magnitude faster. Interestingly, the experiments in [**IV**] exhibit very little correlation between the performance of BMC and explicit-state model checking.

In the above experiments, the total order $\prec$ of the actions is the default order based on the BEEM input file. As different total orders give different sets of allowed steps, the question arises as to which order one should use for best performance. With the serial $\exists$-step semantics, if the control flow graph of a component contains a path consisting of actions, say, $\alpha_1 \to \alpha_2 \to \alpha_3$, then we might prefer the order $\alpha_1 \prec \alpha_2 \prec \alpha_3$ to potentially execute the entire path in a single serial $\exists$-step. A heuristic ordering based on this intuition is applied in [**IV**]. Compared to the default ordering, experiments show differences in solver time of up to an order of magnitude—unfortunately, in both directions. This indicates that search-

ing for a consistently good ordering scheme would be a viable future topic.

With the parallel ∃-step semantics, the choice of the total order is found to play just a minor role on performance. This is because even the worst order permits any set of pairwise independent actions in a parallel ∃-step, and even the best order does not allow any causality between the actions in a step.

5.1.6 Refining Independence

The definitions in the preceding sections allow some flexibility in the notion of independence between ground actions. While the serial ∃-step semantics is immune to any approximation of independence, this affects the set of parallel ∃-steps allowed.

Example 5. Consider a system with an integer array $a[0..2]$, modeled with three state variables a_0, a_1, and a_2, which hold the values $a[0]$, $a[1]$, and $a[2]$. Let ρ be an action that executes the statement $x \leftarrow a[i]$. The value assigned to x is determined by the values of a_0, a_1, a_2, and i, so we can represent the action as the set consisting of all ground actions of the form

$$\langle \rho, \{[i = C], [a_0 = D_0], [a_1 = D_1], [a_2 = D_2]\}, \{x \leftarrow D_C\}\rangle, \tag{5.1}$$

where $C \in \{0, 1, 2\}$ and $D_0, D_1, D_2 \in \mathbb{Z}$. On a more fine-grained level, we see that only one of the array elements is relevant, depending on the value of i. Thus, an alternative representation ρ' for the action consists of the ground actions

$$\langle \rho', \{[i = C], [a_C = D]\}, \{x \leftarrow D\}\rangle, \tag{5.2}$$

where $C \in \{0, 1, 2\}$ and $D \in \mathbb{Z}$. In this form, the action accesses different variables depending on the current state. Consequently, the dependence on other actions becomes conditional on the state. Consider an action σ that contains the assignment $a[0] \leftarrow 0$. The actions ρ' and σ are dependent if $i = 0$ in the current state, and independent otherwise.

In [**III**], steps built on ground actions that follow the form of (5.1) are called static steps, and those built on a state-aware formulation like (5.2) are called dynamic steps. The static over-approximation of variable accesses is always legal and possible, and [**III**] and [**IV**] discuss how it affects the encoding of the parallel ∃-step transition formula. While static steps may result in a simpler formula, the over-approximation of dependence places restrictions on which steps are allowed, and may increase the required bound to find a counterexample.

Steps semantics and queues. There is a limit on how far we can refine independence using ground actions. For example, the statements $x \leftarrow -x$ and $x \leftarrow 2x$ cannot disable each other, and their execution order is irrelevant, so they could be treated as independent. However, we cannot capture such independence because ground actions do not differentiate between commuting and non-commuting assignments.

These considerations are relevant if we combine step semantics and the queue encodings of Sect. 4. Recall that the queue interface of Sect. 4.1 assumes that in a single evolution round of the queue, the accessors $empty^t$, $pending^t$, $full^t$, and $firstelem^t$ are evaluated first, and then zero or more of the dequeue, defer, flush, and enqueue operations are performed in this order. Under the serial $\exists$-step semantics, each of the n actions of the system might in the worst case be able to perform operations on the same queue. Therefore, the serial $\exists$-step transition formula unrolled to bound k needs to be conjuncted with a queue encoding over $\mathcal{O}(kn)$ evolution rounds.

Consider the parallel $\exists$-step semantics and a state variable q, whose type is a (bounded) queue. In terms of ground actions, the queue accessors can be mapped to guards on the value of q, and the queue operations map to a guard and an assignment. For example, a dequeue operation involves a guard $[q = Q]$ and an assignment $q \leftarrow dequeued(Q)$. Thus, this scheme allows parallel $\exists$-steps where a number of actions can use the accessors of the queue, and the last of those actions in the total order can additionally perform queue operations. Such parallel $\exists$-steps can be mapped directly to evolution rounds of the queue, without the need to allocate several rounds of the queue evolution for each step.

More generally, a single queue evolution round could also accommodate a step where one action executes a dequeue operation and another action an enqueue operation. These actions could occur in whichever order, as the operations commute when they both are enabled. Generally, a parallel $\exists$-step could contain a sequence of actions, each reading some of the accessors and performing some queue operations. Some of these sequences could be mapped to a single evolution round at the queue interface, while others could not. Such considerations, however, are more complicated than what can be expressed using ground actions and simple read/write access of state variables. Therefore, this generalization is left out of the scope of publication [**IV**]. Nevertheless, the transition formula of [**III**] implements some progress towards this direction. Specifically, the formula

allows to perform a dequeue and an enqueue operation in a single parallel $\exists$-step, either by a single action or by two separate actions in whichever order. This is implemented only in a restricted setting, where each dequeue operation is preceded by checking the emptiness of the queue and reading the first element, each enqueue operation is preceded by checking that the queue is not full, and any other kind of queue access is not possible.

5.2 Process Semantics

Besides step semantics, publication [**IV**] also investigates a further alternative known as process semantics. The motivation is that while step semantics adds shortcut edges to the state space, it introduces redundancy into the set of executions. In BMC, the multitude of potential counterexample executions to a safety property may slow down satisfiability solving. The idea of process semantics is to reduce the set of allowed executions without increasing the required bound to reach a given state. The principle is that in an execution under process semantics, each action is executed at the earliest opportunity or not at all. Thus, an action cannot be executed at a step if it could have been as well executed at an earlier step.

Any state reachable by a step semantics execution of exactly k steps is also reachable by a process semantics execution of k or fewer steps. On the other hand, as process semantics does not add anything to the set of allowed executions, the minimum bound to reach a given state is not reduced with respect to step semantics. Like step semantics, process semantics preserves invariant properties with respect to the interleaving semantics.

In [**IV**], we present a new variant of process semantics called the *serial process semantics*, which builds on the serial $\exists$-step semantics. Recall that a serial $\exists$-step execution consists of an initial state and a finite sequence of steps of the form $(g^1 \dots g^n)$, where each g^i is a ground action in $gnd(\mathsf{act}^i) \cup \{\text{-}\}$, and there is at least one i such that g^i is not the skip ground action -. To simplify the presentation, in this section we disregard the internal representation of ground actions. Instead, we treat ground actions abstractly as symbols of an alphabet with an independence relation. Each action act^i is represented as a set of symbols $A^i = gnd(\mathsf{act}^i)$.

Definition 5. Assume a disjoint union of sets of symbols $A^1 \cup \cdots \cup A^n \cup \{\text{-}\}$, where - is the *skip symbol*, and a binary, symmetric, irreflexive *independence* relation I over the symbols such that - is independent of any other symbol and no two symbols in A^i are independent for any i. A *step* is a string of the form $(g^1 \dots g^n)$, where each g^i is in $A^i \cup \{\text{-}\}$. The step $(\text{-} \dots \text{-})$ consisting of only n skip symbols is the *empty step*. A *step sequence* is a finite sequence of steps. The *linearization* of a step sequence is the string obtained by dropping all grouping parentheses and skip symbols from the sequence.

Example 6. Let our system contain the actions $\mathsf{act}^1 \prec \mathsf{act}^2 \prec \mathsf{act}^3 \prec \mathsf{act}^4$ represented by symbol sets A^1, A^2, A^3, and A^4, and let

$$(\text{-}g_2\text{-}\,\text{-})(g_1\,\text{-}\,\text{-}h)(\text{-}\,\text{-}g_3 g_4) \tag{5.3}$$

be a step sequence with symbols $g_1 \in A^1$, $g_2 \in A^2$, $g_3 \in A^3$, and $g^4, h \in A^4$ such that h is independent of g_1, g_2, and g_3, while all other pairs of symbols are dependent. By breaking the steps down to individual unit steps, we get another step sequence

$$(\text{-}g_2\text{-}\,\text{-})(g_1\,\text{-}\,\text{-}\,\text{-})(\text{-}\,\text{-}\,\text{-}h)(\text{-}\,\text{-}g_3\,\text{-})(\text{-}\,\text{-}\,\text{-}g_4). \tag{5.4}$$

The sequences (5.3) and (5.4) have the same linearization $g_2 g_1 h g_3 g_4$. Reordering independent ground actions does not affect the state reached by executing them. This principle allows switching the order of h and the adjacent symbols in the linearization, yielding the strings $g_2 g_1 g_3 h g_4$ and $g_2 h g_1 g_3 g_4$. These strings can be seen as linearizations of the step sequences

$$(\text{-}g_2\text{-}\,\text{-})(g_1\,\text{-}g_3 h)(\text{-}\,\text{-}\,\text{-}g_4) \tag{5.5}$$

and

$$(\text{-}g_2\,\text{-}h)(g_1\,\text{-}g_3 g_4), \tag{5.6}$$

respectively. The step sequences (5.3)–(5.6) are all equivalent in terms of Mazurkiewicz traces, formalized below. That is, the linearizations of the sequences can be obtained from one another by repeatedly transposing adjacent independent ground actions.

Definition 6. *Mazurkiewicz traces* [66]. Assume a set Σ of symbols and a symmetric, irreflexive independence relation I over the symbols. Two finite strings w, w' over Σ are *trace equivalent*, denoted by $w \equiv_\Sigma w'$, if and only if there is a finite sequence of strings $w_0, \dots, w_N$ such that $w = w_0$, $w' = w_N$, and for all $i = 1, \dots, N$, there are strings u, v and symbols $a, b \in \Sigma$

such that $(a, b) \in I$, $w_{i-1} = uabv$, and $w_i = ubav$. Equivalence classes of the relation $\equiv_\Sigma$ are called *traces*.

Definition 7. The trace of a step sequence x, denoted by $trace(x)$, is the trace that contains the linearization of x.

In the example above, all step sequences (5.3)–(5.6) have the same trace. Generally, if a set of executions have the same initial state and their sequences of steps have the same trace, then they also reach the same final state. If this is the case, we would only need to consider one of the executions when model checking safety properties.

In Example 6, only the last step sequence (5.6) fulfills the principle of process semantics, which is to execute each action as early as possible. We say that the step sequence is in the *serial process normal form*, defined below. The characterizing property is that no symbol can be pushed back to an earlier step because there is always an interfering dependent symbol that prevents the reordering.

Definition 8. A step sequence $(g^{0:1} \dots g^{0:n})(g^{1:1} \dots g^{1:n}) \cdots (g^{k-1:1} \dots g^{k-1:n})$ is in the *serial process normal form* iff it contains no empty step, and for all $1 \leq t \leq k-1$ and $1 \leq i \leq n$ such that $g^{t:i} \neq$ -, the symbol $g^{t:i}$ is dependent on at least one of the n preceding symbols $g^{t-1:i}, \dots, g^{t-1:n}, g^{t:1}, \dots, g^{t:i-1}$.

Example 7. The step sequence $(\text{-}g_2\text{--})(g_1\text{--}h)(\text{--}g_3g_4)$ in Example 6 is not in the serial process normal form because the instance of g_3 is independent of the preceding 4 symbols $(\text{-}h\text{--})$. Thus, we can transform the sequence to $(\text{-}g_2\text{--})(g_1\text{-}g_3h)(\text{---}g_4)$, which has the same trace but is closer to the serial process normal form in the sense that one of the symbols occurs earlier. Repeating such transformations eventually leads to the step sequence (5.6), which is in the serial process normal form.

In bounded model checking with the serial process semantics, the idea is to disregard all executions not in the serial process normal form. We unroll the serial $\exists$-step transition formula as usual. In the unrolled formula, we add constraints that enforce the serial process normal form. That is, for every ground action in $gnd(\mathsf{act}^i)$ executed from the second step onwards, there has to be either a dependent ground action in $gnd(\mathsf{act}^j)$ with $j < i$ executed at the same step or a dependent ground action in $gnd(\mathsf{act}^j)$ with $j \geq i$ executed at the previous time step. Recall that dependence is defined in terms of the sets of state variables written and read by the ground actions. Therefore, the new constraints are built on time-

dependent formulas that track whether each state variable has been written or read by a recent action. The interpretations that satisfy the resulting formula then correspond to serial $\exists$-step executions in the serial process normal form. Full details are in [**IV**].

5.2.1 Analysis of the Serial Process Normal Form

Publication [**IV**] leaves open the question whether the serial process normal form yields canonical representatives of Mazurkiewicz traces. The answer is positive. Of course, such a result is subject to the exact mapping of executions to strings and the definition of independence.

Theorem 1. *Under the assumptions of Definitions 5–8, for any trace M over the symbols $A^1 \cup \cdots \cup A^n$, there exists exactly one step sequence x such that $trace(x) = M$ and x is in the serial process normal form.*

Proof. Let $X := trace^{-1}(M)$ be the set of step sequences whose linearization is in the trace M. Let us first show that X contains at most one step sequence in the serial process normal form. Assume that

$$x = (g^{0:1} \dots g^{0:n})(g^{1:1} \dots g^{1:n}) \cdots (g^{k-1:1} \dots g^{k-1:n}) \quad \text{and}$$
$$y = (h^{0:1} \dots h^{0:n})(h^{1:1} \dots h^{1:n}) \cdots (h^{m-1:1} \dots h^{m-1:n})$$

are different step sequences with $trace(x) = trace(y) = M$. If one of x and y is a prefix of the other, then the longer sequence must have an empty step in the end and is thus not in the serial process normal form. Otherwise, let the first point of difference of the two sequences be $g^{t:i} \neq h^{t:i}$. At least one of $g^{t:i}$ and $h^{t:i}$ is not the skip symbol, say, $h^{t:i} \neq$ -. Because $trace(x) = trace(y)$, there is an index $u > t$ such that $g^{u:i} = h^{t:i}$ and $g^{u:i}$ is independent of the symbols of x between $g^{t:i}$ (inclusive) and $g^{u:i}$ (exclusive). In particular, $g^{u:i}$ is independent of the n preceding symbols, thus x is not in the serial process normal form. This proves the first part.

Then, we will show that X contains at least one sequence in the serial process normal form. As an auxiliary concept, define the weight of a step sequence $x = S_0 \cdots S_{k-1}$ by

$$w(x) = 2^k + \sum_{t=0}^{k-1} \frac{|S_t|}{n} 2^t,$$

where $|S_t|$ denotes the number of non-skip symbols in step S_t. In particular, $|S_t|$ is between 0 and n, thus $2^k \leq w(x) < 2^{k+1}$.

If we take an arbitrary string $g_0 \dots g_{k-1}$ in M and turn each g_t into the corresponding unit step, we get a step sequence in X. Thus, the set X is

nonempty. Take from X a step sequence with minimum weight and call it $x = (g^{0:1} \dots g^{0:n})(g^{1:1} \dots g^{1:n}) \cdots (g^{k-1:1} \dots g^{k-1:n})$. Then x cannot contain an empty step; if it did, we could drop the empty step from x to obtain a sequence in X with a lower weight. Let us show that x is in the serial process normal form. Assume that this is not the case. Then, there exist $1 \leq t \leq k-1$ and $1 \leq i \leq n$ such that $g^{t:i} \neq$ - and $g^{t:i}$ is independent of the symbols $g^{t-1:i}, \dots, g^{t-1:n}, g^{t:1}, \dots, g^{t:i-1}$. As no two symbols in A^i are independent, we must have $g^{t-1:i} =$ -. Let y be the sequence obtained from x by switching $g^{t-1:i}$ and $g^{t:i}$, i.e. by moving the symbol $g^{t:i}$ to the preceding step. As this transformation only involves transpositions of independent symbols, $trace(y) = trace(x)$, and y is in X. However, $w(y) < w(x)$, which contradicts the choice of x. The assumption that x is not in the serial process normal form is thus false. This concludes the proof. □

Given a step sequence in the serial process normal form, there is no shorter step sequence whose linearization is in the same trace. Therefore, switching from the serial ∃-step semantics to the serial process semantics does not increase (nor decrease) the minimum bound required to reach a given state.

Theorem 2. *Under the assumptions of Definitions 5–8, if x and y are step sequences such that $trace(x) = trace(y)$, and x is in the serial process normal form, then y contains at least as many steps as x.*

Proof. Continuing the previous proof, if k is the number of steps in x, we have $2^k \leq w(x) \leq w(y)$. If y contained fewer than k steps, we would have $w(y) < 2^k$, a contradiction. □

5.2.2 Experiments with Process Semantics

In [**IV**], the serial ∃-step semantics and the serial process semantics are compared experimentally on the BEEM benchmark set. The introduction of the process constraints in the BMC formula is shown to increase the formula size by a linear factor. According to measurements, the size roughly doubles in practice. Figure 5.4(a) shows the effect on the total solver time to find counterexample executions to invariant properties. Recall that the required minimum bound is the same with the two semantics. It seems that the extra constraints generally neither help nor hinder finding a counterexample. Figure 5.4(b) displays the cases where neither approach could find a counterexample. Each marker denotes the cumulative solver

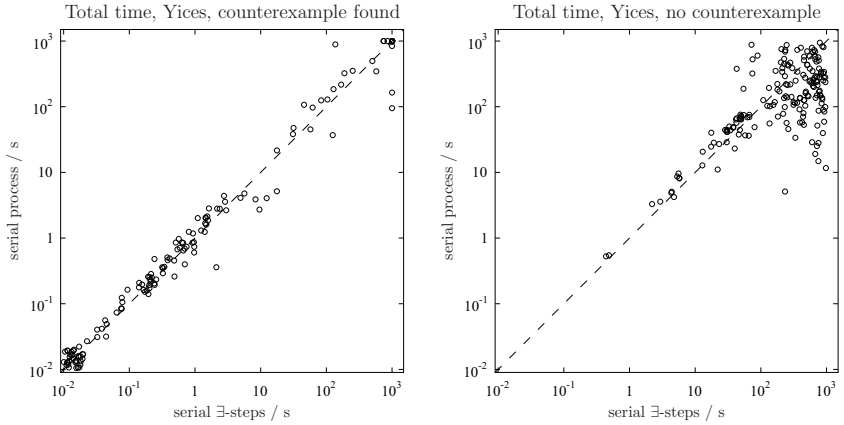


Figure 5.4. BMC efficiency with the serial ∃-step vs. serial process semantics transition formulas.

time up to a bound that was reached by both approaches within the time limit. Although the effect is generally bilateral, the process constraints in most cases speed up the refutation of the unrolled BMC formula, at least with the Yices 2.0 solver [93] used in these experiments.

5.3 Bounded Event Tracing

Publication [**V**] introduces *bounded event tracing*, a new framework for checking bounded safety properties of concurrent systems by reduction to satisfiability problems. It is a variant of SAT-based bounded model checking, but is given another name because it is not based on unrolling a transition formula. Instead, the bound is set by defining a finite set of potential events such that each potential event can be executed at most once, and such executions map to a bounded portion of the behavior of the system. The structure that contains the potential events is called an *unwinding* of the system. Technically, an unwinding is a high-level Petri net whose transitions are the potential events. To cover executions of the system where an action occurs more than once, the unwinding needs to have several transition instances for each action. Increasing the bound is done by creating a new unwinding with more transitions. Bounded event tracing, like BMC, finds counterexample executions but is not directly capable of proving safety properties.

The difference to BMC is in how the unwinding is translated to a for-

mula that characterizes the bounded set of executions. Whenever a transition reads a piece of data or a control token, that precondition of the transition must have been previously produced by an earlier transition. Instead of a number of fixed time steps, the formula directly describes the nondeterministic choice of where the data or control token came from. Realization of this choice creates causal links between the transitions. No ordering of the transitions is imposed other than that induced by these links. In the presence of concurrency, the links only induce a partial execution order. Therefore, bounded event tracing completely avoids the problem caused by different interleavings of independent occurrences. Another aspect of BMC is that the presence of fixed, global time steps may introduce spurious synchronization between components, even with alternative execution semantics. Bounded event tracing, on the other hand, has no global time steps but only local ordering of interacting transitions.

Publication [**V**] defines the structure and semantics of unwindings and presents an automatizable translation to formulas that are given to an SMT or SAT solver. Correctness proofs are in the technical report version [34]. The method is immature in that a robust way to construct efficient unwindings for a system is still missing. As a proof of concept, [**V**] presents a unwinding scheme that covers many of the features used in the models of the BEEM benchmark set [80]. In the following, we take an overview of these aspects.

5.3.1 A Model Checking Procedure

The proposed procedure for checking safety properties with bounded event tracing is as follows. We start with a concurrent system with discrete execution steps and a safety property. We map the actions of the systems to potential events such that finite executions up to some bound are covered by executing each potential event at most once. A single action may map to several potential events. The safety property is mapped to a special potential event $t^\diamond$ such that executing $t^\diamond$ violates the property. Then, we form an unwinding as a Petri net whose transitions are the potential events. Using the automatic translation defined in [**V**], we construct a formula from the unwinding and check the satisfiability using an off-the-shelf SMT solver. If the formula is satisfiable, we extract an execution that falsifies the safety property. If the formula is unsatisfiable, we increase the bound by adding more potential events and start over with a

bigger unwinding.

With this scheme, there is no simple way to verify an unbounded safety property if the property holds.

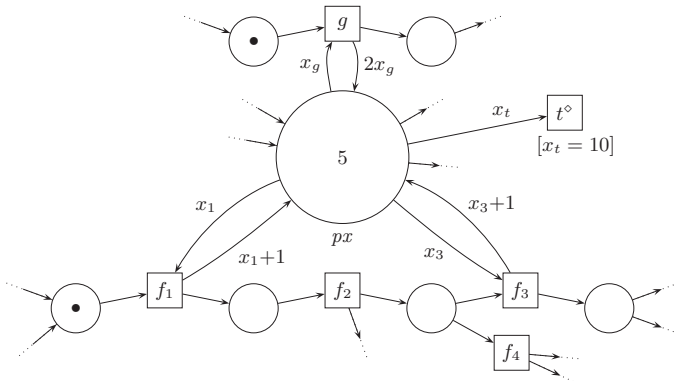
5.3.2 Structure and Semantics of Unwindings

An unwinding is a Petri net that describes the potential events and their interactions. It is a high-level Petri net [54], which means that its transitions can manipulate data values. Unwindings respect the execution semantics of Petri nets, with the crucial restriction that each transition may be executed at most once. Therefore, an unwinding is not a complete model of a system but only specifies a bounded portion of the system behavior. Repetition in the behavior cannot be expressed as a cycle in the unwinding. Instead, repetition is modeled by making new copies of transitions. Although the length of executions is bounded by the number of transitions, an unwinding may have infinitely many reachable states due to nondeterministic choices over infinite domains.

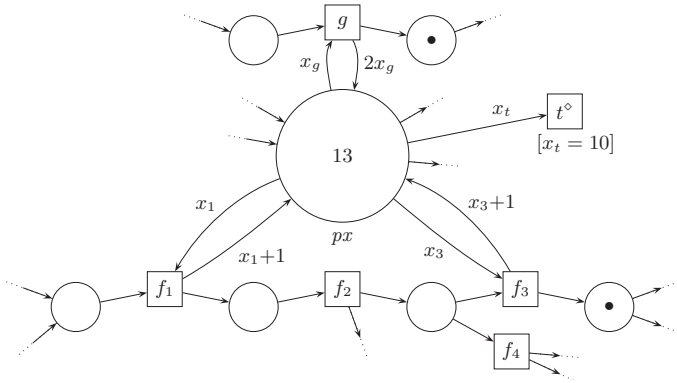
We will go through these concepts with the help of an example. Formal definitions are found in [**V**], as well as a more detailed explanation of the graphical notation. Figure 5.5(a) shows a part of an unwinding. The rectangles and circles denote *transitions* and *places*, respectively. The place px models a shared integer variable x, while the other places model control locations of system components. The markers $\bullet$ and 5 inside the places are not part of the unwinding but denote *tokens*. Generally, a *state* of an unwinding consists of a multiset of tokens in each place. Each token carries a value. A token may also be a pure control token and carry the meaningless value in the singleton domain $\{\bullet\}$. Once the unwinding has been constructed, the method does not differentiate between control and data tokens but treats them in a unified way.

When a transition is *executed* atomically, it consumes a token with each of its associated *input arcs* and produces a token with each of its *output arcs*. The arcs are denoted by arrows labeled with expressions that specify the value of the consumed or produced token. A transition is *enabled* if all its input arc expressions are satisfied by suitable tokens and its *guard* predicate is satisfied. If a transition has a guard, it is denoted by a Boolean expression in square brackets.

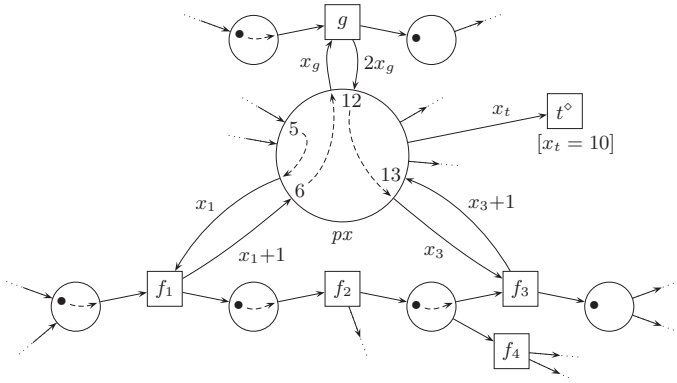
This is the standard structure and semantics of Colored Petri Nets [54], except that we only allow each arc to consume or produce exactly one



(a)



(b)



(c)

Figure 5.5. Part of an unwinding in different states (a) and (b). In (c), the token values and links (dashed arrows) inside the places (circles) denote a particular token trace of the unwinding.

token at a time.

Figure 5.5(a) illustrates the outcome of one possible scheme for constructing unwindings. The control flow of each component is transformed to a directed acyclic graph of transitions and places. In this process, each loop is unwound up to a bound. In the example, a control token can flow through transitions f_1 and f_2 to a branch where either f_3 or f_4 is executed. Each shared variable is modeled using a single place and input/output arc pairs for write access. For example, transition f_1 models an action that updates variable x by consuming its old value and producing a new value $x + 1$. The transition $t^\diamond$ is special in that it models a violation of the safety property. In this case, the property is the invariant $x \neq 10$, and $t^\diamond$ is enabled in every state that breaks the invariant. A more detailed explanation of this unwinding scheme is in [**V**]. Also, [**V**] extends unwindings with *test arcs* [20], which read the value of a token without consuming it, useful for modeling read-only access to shared variables.

Figure 5.5(b) shows the state reached by starting from Fig. 5.5(a) and executing the transitions f_1, f_2, g, and f_3. Generally, an *execution* of an unwinding consists of a sequence of transitions and concrete values for the produced tokens. According to the restriction to boundedness, we only consider *one-off executions*, which are executions where each transition occurs at most once. We assume that the construction of the unwinding is such that its one-off executions map easily to finite executions of the system.

Token traces. Checking the bounded safety property reduces to deciding the existence of a one-off execution of the unwinding in which $t^\diamond$ occurs. However, to reflect the partial order view of behavior in the model checking process, we do not search for one-off executions directly. Instead, we look for a *token trace* of the unwinding. Like a one-off execution, a token trace defines a set of *events* that occur, i.e. a subset of the transitions, and concrete values for tokens consumed and produced by the associated arcs. Instead of fixing a linear order of events, a token trace only specifies a partial order by associating each input arc of each event to an output arc of another event. These associations are called *links* and they describe the flow of tokens. Figure 5.5(c) depicts a token trace that corresponds to the execution described earlier.

A token trace can be mapped to a corresponding one-off execution, or

generally, to a set of one-off executions, by taking any linearization of the partial order of events induced by the links. For example, the token trace in Fig. 5.5(c) can be mapped to a one-off execution where f_2 occurs before g, or vice versa. In the other direction, every one-off execution of an unwinding can be mapped to a token trace. This mapping is not one-to-one either if it happens that at some point in the execution, a place contains several identical tokens and a transition consumes one of them. This is because a token trace identifies the output arc that produced the token being consumed, while a one-off execution does not.

5.3.3 Encoding Token Traces

For symbolic model checking, [**V**] presents a mechanical translation from an unwinding to a formula that characterizes the token traces of the unwinding. A satisfiability solver is then used to check the existence of an interpretation that corresponds to a token trace where $t^\diamond$ is an event.

The formula encodes the rules for a valid token trace. In particular, it is required that each input arc of each event is linked to an output arc, and no two input arcs are linked to the same output arc. The token value consumed by each input arc must be the same as the value produced by the linked output arc. A crucial constraint is that the links in a token trace must not induce a causal cycle. For example, there can be no valid token trace in which transition f_1 in Fig. 5.5 consumes from place px a token produced by f_3. This acyclicity is enforced by constraints expressed in *difference logic* over real or integer variables—in this case, using only subformulas of the form $x < y$. Difference logic is supported by many SMT solvers, e.g. Yices [93]. All these constraints are expressed as constraints local to each place. In particular, the occurrence of transitions is not tied to fixed time steps. Because any input arc incident to a place is potentially linked to any output arc incident to the same place, the size of the encoding of each place is quadratic in the number of arcs incident to the place. In the presence of test arcs, the encoding is cubic in the worst case. Furthermore, it is possible to encode the acyclicity constraints, instead of difference logic, by an eager SAT encoding, whose size is cubic in the number of transitions [34].

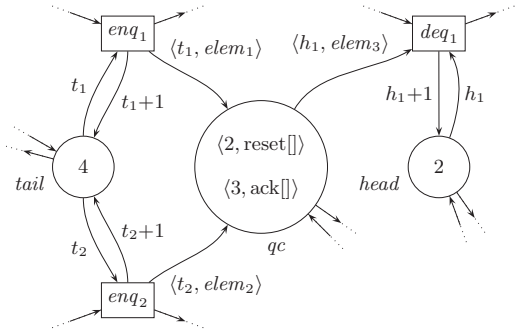


Figure 5.6. An unwinding that illustrates one way to model a queue. The token values inside the places (circles) are not part of the unwinding, but represent a state where the queue contains the two elements $\langle \text{reset}[], \text{ack}[] \rangle$.

5.3.4 Representing Queues

Besides shared variables, unwindings may need to model other forms of communication between concurrent components. Regarding communication through a first-in-first-out queue, the queue encodings of Sect. 4 are not directly usable in bounded event tracing. Figure 5.6 shows one possible way to represent in an unwinding an unbounded queue without deferring capability.

The solution is similar in spirit to the linear queue encoding approach in Sect. 4.2.3. Transitions enq_1 and enq_2 represent actions that perform an enqueue operation, while transition deq_1 performs a dequeue operation and also reads the dequeued element. The place qc holds the current contents of the queue as a multiset of tuples of the form $\langle index, elem \rangle$. The places $tail$ and $head$, initialized with tokens of value 0, hold the integer index of the next element to be enqueued and dequeued, respectively.

5.3.5 Relation to Alternative Execution Semantics

A natural question is how bounded event tracing relates to bounded model checking with interleaving or alternative execution semantics. In BMC, unrolling the transition formula is purely mechanical and results in a sequence of homogeneous time steps, where a copy of each action is encoded for each time step. In contrast, the construction of unwindings allows a great deal of flexibility; in particular, different actions of the system may map to different numbers of transitions in an unwinding. A potential bottleneck in bounded event tracing is that making duplicated copies of

transitions increases the number of arcs incident to each place, causing bloat in the SMT formula. A sophisticated unwinding scheme might take this into account and judiciously make copies of places as well.

As discussed in Sect. 5.2, the serial process semantics is a partial order semantics that captures exactly one execution from each Mazurkiewicz trace (Definition 6). Like a Mazurkiewicz trace, a token trace also maps to a set of executions equivalent up to a partial order. However, we cannot immediately conclude that token traces match Mazurkiewicz traces. Consider token traces of the unwinding in Fig. 5.6. A token trace that links deq_1 to enq_2 does not necessarily induce any ordering between enq_1 and deq_1. However, an occurrence of deq_1 is generally not independent of enq_1, and therefore a Mazurkiewicz trace over strings of transition occurrences must put enq_1 and deq_1 in one order or the other. In this sense, a token trace is able to cover executions from several Mazurkiewicz traces. On the other hand, token traces, unlike strings of transition occurrences, may superfluously differentiate between identical tokens if they reside at the same time in the same place. Apparently, we might be able to obtain one-to-one correspondence between token traces and Mazurkiewicz traces if we represented executions as words of an alphabet where each symbol identifies not only a transition and the data values, but also the output arcs from which the consumed tokens originate. In this work, we will not however elaborate this idea further.

5.3.6 Experiments with Bounded Event Tracing

In [**V**], bounded event tracing is experimentally compared to bounded model checking with an interleaving transition formula on four different BEEM benchmarks [80]. The construction of unwindings is based on fixing a loop bound L and making L unwound copies of loops in the control flow graphs of the components. The safety property is checked for $L = 0, 1, 2, \ldots$ until a counterexample is found or resources are exhausted. On some benchmarks, bounded event tracing clearly outperforms bounded model checking by covering a larger portion of the reachable state space in a given amount of solver time. On other instances, bounded event tracing overburdens the SMT solver much more rapidly than BMC. The probable reason is the unwieldy growth in the formula size due to the primitive unwinding scheme that may generate hundreds of transitions that are never enabled.

5.4 Related Work

Partial order reduction methods for explicit-state model checking prune unnecessary states or transitions from the state space under the guarantee that the property to be checked is unaffected. A comprehensive overview is given in Godefroid's dissertation [44]. The sole purpose of partial order reduction is to increase performance. For symbolic model checking, such pruning is not necessarily the best strategy because symbolic model checkers do not work by enumerating reachable states. In the following, we will review proposed symbolic methods for mitigating the state explosion problem caused by interleavings.

Interleaving semantics with reductions. An approach called peephole partial order reduction [96], like our step semantics approach, builds on BMC with a transition formula that implements the interleaving semantics. The reduction is implemented by adding a constraint that each pair of independent actions can occur at consecutive time steps only in one predefined order. A generalization called monotonic partial order reduction [59] claims to explore exactly one execution for each Mazurkiewicz trace. Another approach [46] is to start BMC with an under-approximation that allows a limited set of interleavings, and then allow more interleavings by iteratively removing constraints.

Step semantics. Our parallel $\exists$-step semantics is most closely related to the SAT-based planning approach by Rintanen et al. [86]. While the actions in our formalism handle data values from arbitrary, abstract domains, the planning operations employed in [86] operate on only Boolean data. Accordingly, the dependency relationship between operations that access the same variable is conditioned to the written value (true or false). The relaxed $\exists$-step planning approach by Wehrle et al. [97] falls between parallel and serial $\exists$-steps in that all operations in a step are not required to be enabled in the beginning of the step, but variables can still change their value at most once per step.

The serial $\exists$-step semantics corresponds to the idea of the BMC approach that Ogata et al. [77] presented for 1-safe Petri nets.

Process semantics. The idea of process semantics was introduced by Heljanko [48] for 1-safe Petri nets and applied to labeled transition systems by Jussila [56]. The process semantics in those works requires executions to be in the Foata normal form, discussed in [48]. The Foata normal form consists of a sequence of steps such that (i) the actions in a step are pairwise independent, and (ii) every action in each step from the second step onwards is dependent on at least one action in the preceding step. In particular, such executions conform to the $\forall$-step semantics, which in turn is subsumed by the serial $\exists$-step semantics (Fig. 5.1). As discussed in Sect. 5.2.1, each serial $\exists$-step execution is Mazurkiewicz trace equivalent to a serial process execution with the same or lower number of steps. It follows that our serial process semantics yields executions at least as compact in the number of steps as the process semantics in [48, 56].

Component-wise BMC with external synchronization. There is a range of BMC techniques with the common basis that instead of unrolling a transition formula for the whole system, the concurrent components are first treated in separation. This circumvents the problem of nondeterministic interleaving and typically enables simplifications based on the control flow graph of each component. After unrolling the behavior of each component, extra synchronization constraints are added to nondeterministically link the globally visible actions of different components to each other to produce valid executions of the whole system. Techniques in this category include [18, 41, 83]. These works are discussed in more detail in [**V**]. Along similar lines, Bu et al. present a BMC approach [16] where each component proceeds individually on its local (continuous) time scale, and constraints are placed to align the clocks on each synchronizing action.

One could say that bounded event tracing also falls into this category of techniques. Although unwindings as a generic framework are oblivious to the concept of concurrent components, a sensible unwinding scheme unrolls the control flow of each component separately and adds extra places and arcs to model the communication between components.

Unfoldings. A completely different SAT-based technique for concurrent systems is based on unfoldings [67, 36], which are partial order representations of state spaces as acyclic low-level Petri nets. Although an unfolding represents interleavings implicitly, every possible control path is ex-

plicitly present. Thus, unlike with BMC, the generation of (finite prefixes of) unfoldings is the most expensive part, not the SAT solving. Unlike unwindings (Sect. 5.3), unfoldings cannot represent data symbolically.

Merged processes [61] diminish the problem of diverging paths in unfoldings. Despite the apparent similarities between our unwindings and merged processes, the techniques of bounded event tracing and merged processes are fundamentally different. In particular, merged processes restrict not only each transition but also each place to be used at most once during an execution.

5.5 Discussion

This section and publications [**III**], [**IV**], and [**V**] present symbolic partial order methods to accelerate model checking of concurrent systems.

The presented parallel ∃-step semantics and especially the serial ∃-step semantics are found to significantly speed up bounded model checking. A step semantics transition formula works, to a large extent, as a drop-in replacement for an interleaving transition formula. This makes step semantics immediately applicable to bounded model checking of safety properties—however, more complex temporal properties would require special treatment because the shortcut edges in the state space may miss intermediate states. In particular, the parallel ∃-step semantics has been implemented on top of the transition formula for hierarchical UML state machines in the SMUML toolset [92]. Step semantics also works with BDD-based model checking, although the effect on performance may be negative, as indicated by the experiments in [**III**]. A particularly interesting research direction would be to combine step semantics and the complete BMC approach based on Craig interpolants [68].

The novel serial process semantics refines the serial ∃-step semantics with the attractive property of allowing only one interleaving for each Mazurkiewicz trace. The applicability of process semantics in settings other than plain BMC may be complicated by the fact that the extra constraints need to refer to two subsequent steps and thus cannot be expressed in a simple way within a transition formula.

The third partial order approach, bounded event tracing, breaks out of the view of executions as linear sequences. Bounded event tracing offers a clean, unified framework—conveniently visualized as Petri nets—

for SMT-based checking of safety properties, but it is not at this point clear how to make maximal use of the framework. An interesting question is whether the performance benefits of bounded event tracing can surpass step and process semantics. More practical experimentation is needed to answer the question. A major challenge is in finding ways to guide the construction of unwindings to cover as much behavior as possible without suffering from the superlinear growth in the formula size.

6. Structure-Aware Predicate Abstraction

A key technique in tackling the state explosion problem is the application of abstract interpretation [31] to disregard details that are irrelevant with respect to the system property under inspection. *Predicate abstraction* [45] has lately gained popularity due to its ability to automatically map a large or infinite state space to an abstract model, whose states are Boolean vectors, thus enabling to leverage symbolic model checking on the abstract model. Publication **[VI]** addresses a practical challenge of predicate abstraction: computing the abstract model is generally expensive and can present a bottleneck in the model checking process.

Predicate abstraction is an instance of *existential abstraction* [25], where the abstract model is defined by an *abstraction function* α that maps concrete states to abstract states. In the abstract state space, there is a transition from an abstract state a to a' if and only if there *exists* a concrete transition $s \rightarrow s'$ with $s \in \alpha^{-1}(a)$ and $s' \in \alpha^{-1}(a')$. This conservative abstraction ensures that if the abstract model satisfies the safety property "no state in a set A is reachable", then the concrete model satisfies the corresponding concrete property "no state in $\alpha^{-1}(A)$ is reachable". In predicate abstraction, the abstraction function is determined by a finite set of *abstraction predicates* on states. Two concrete states s_1 and s_2 map to the same abstract state if and only if $\gamma(s_1) \leftrightarrow \gamma(s_2)$ for all abstraction predicates γ.

If an abstract model is too weak to verify the desired property, it needs to be refined to more closely resemble the concrete system. In predicate abstraction, this refinement can be done by adding new abstraction predicates. A key innovation is Counterexample-Guided Abstraction Refinement (CEGAR [24]), which consists of iterations where an abstract model is computed, model checking is applied on the abstract model, and if the property does not hold in the abstraction, a counterexample execution is

analyzed for extracting new predicates. With this process, CEGAR automatically and iteratively adapts the abstract model to the problem.

To enable symbolic (typically BDD-based) model checking of the abstract model, we need construct its transition formula as a propositional formula over the abstract Boolean state variables. This is an instance of a quantifier elimination problem, where the concrete state variables are existentially quantified. According to experience, constructing the abstract transition formula is often far more expensive than any other phase in the CEGAR process [98]. To leverage solver technology, enumerative approaches based on All-SAT and All-SMT [64] have been proposed. So far, these approaches have taken a monolithic view of the system. In [**VI**], we suggest instead to partition the problem to smaller subproblems, and to apply a quantifier elimination procedure to each subproblem in sequence. Because of the exponential complexity of eliminating a quantifier from a formula, this divide-and-conquer approach can bring significant performance benefits. The partitioning follows the structure of the system model.

In [**VI**], this idea is instantiated to the system formalism of linear hybrid automata networks [1, 3], which consist of communicating state machines with continuous-time behavior. In the following sections, we will skip the definition of hybrid automata, and introduce the structural abstraction concepts on a higher level. Section 6.1 presents the problem of computing predicate abstractions, and Sect. 6.2 goes through our proposed approach. Experiments on hybrid automata are presented in Sect. 6.3, with related work in Sect. 6.4 and concluding discussions in Sect. 6.5.

6.1 Computing Predicate Abstractions

We assume that a finite set of abstraction predicates $\{\gamma_1, \ldots, \gamma_m\}$ is given. The abstract state variables $p_1, \ldots, p_m$ are Boolean variables that correspond to the predicates. A state in the abstract model is a Boolean vector that gives values to the abstract state variables. The abstraction function from concrete states to abstract states is defined by

$$\alpha(s) = \langle \gamma_1(s), \ldots, \gamma_m(s) \rangle.$$

Under existential abstraction, an abstract state a is initial iff there is a concrete initial state s such that $\alpha(s) = a$. There is a transition in the abstract state space from a to a' iff there is a concrete transition $s \to s'$

such that $\alpha(s) = a$ and $\alpha(s') = a'$. If T is a concrete transition formula, the abstract transition formula over abstract states $a = \langle p_1, \ldots, p_m \rangle$ and $a' = \langle p'_1, \ldots, p'_m \rangle$ is equivalent to

$$R(a, a') \equiv \exists s, s' : T(s, s') \wedge C_\Gamma(s, s', a, a'), \tag{6.1}$$

where the *abstraction constraint* C_Γ is defined by

$$C_\Gamma(s, s', a, a') := \bigwedge_{1 \le j \le m} (p_j \leftrightarrow \gamma_j(s)) \wedge (p'_j \leftrightarrow \gamma_j(s')). \tag{6.2}$$

To apply symbolic (BDD-based) model checking to the abstract model, we need to express R as a propositional formula over $p_1, \ldots, p_m, p'_1, \ldots, p'_m$. With this formulation, the predicate abstraction problem is thus reduced to quantifier elimination. The abstract initial state formula is similar but is much simpler to compute and will not be discussed further. The same applies to abstracting the invariant property.

6.1.1 Precise and Approximate Abstraction

Existential abstraction can be implemented either precisely or approximately. Abstraction as such is already an approximation of concrete behavior, but by approximate abstraction we mean allowing the abstract model to have a transition from an abstract state a to a' even though there is no concrete transition from any state in $\alpha^{-1}(a)$ to any state in $\alpha^{-1}(a')$. The reason why approximate abstraction is often applied in practice is that it can be significantly cheaper to compute than the precise abstraction (6.1).

A common approximation is *Cartesian abstraction* [8, 25, 55]. It is a divide-and-conquer approach where, roughly speaking, the effects of a transition on different state variables are abstracted separately, and the results of these cheap sub-problems are joined to form the abstract model. We can express the idea as a general formula-level approximation

$$\exists xyz : f(a, x, z) \wedge g(a, y, z) \;\equiv \tag{6.3}$$

$$\exists xyzz' : f(a, x, z) \wedge g(a, y, z') \wedge (z = z') \;\approxeq \tag{6.4}$$

$$\exists xyzz' : f(a, x, z) \wedge g(a, y, z') \;\equiv \tag{6.5}$$

$$\big(\exists xz : f(a, x, z)\big) \wedge \big(\exists yz' : g(a, y, z')\big), \tag{6.6}$$

where (6.3) represents the original abstraction problem and (6.6) is its Cartesian approximation. From the intermediate forms (6.4) and (6.5),

we see that Cartesian abstraction effectively allows the shared vector of variables z to have different values in the conjuncts f and g.

The price of approximate abstraction is that it may introduce spurious behavior in the abstract model. In the context of predicate abstraction and CEGAR, this means that adding predicates is no longer sufficient for eliminating spurious counterexamples. Additional refinement iterations are needed to remove sets of spurious transitions from the abstract state space. In this work, we focus on precise abstraction, which avoids this extra complexity in the CEGAR loop.

6.1.2 SMT-Based Enumeration

One of our baselines is the piecewise construction of the abstract transition formula by iterative satisfiability checking. Assume that R_0 is an under-approximation of the abstract transition formula, i.e. $R_0 \to R$ is valid. Then, by (6.1), the formula

$$\neg R_0(a, a') \wedge T(s, s') \wedge C_\Gamma(s, s', a, a') \tag{6.7}$$

is unsatisfiable if and only if $R_0 \equiv R$. If the formula is satisfiable by some interpretation, then we can extract concrete values A, A' for the abstract state variables a, a' and augment R_0 to form a new under-approximation

$$R_1 := R_0 \vee \big((a = A) \wedge (a' = A')\big),$$

which includes the new abstract transition $A \to A'$. Starting from an identically false transition formula, this process is iterated until the abstract transition formula is complete. The procedure terminates because in each iteration, at least one new satisfying interpretation is found, and there are only finitely many interpretations of the Boolean vectors a, a'.

Generally, a quantifier elimination problem $\exists x : f(a, x)$, where a is a vector of Boolean variables, can be solved by enumerating the satisfying interpretations restricted to a. This idea for predicate abstraction is employed in [64] using an incremental SMT solver. An alternative presented in [19] uses BDDs instead of the search procedure in an SMT solver to guide the construction of the abstract transition formula.

6.1.3 Hindrances to Structural Simplification

The SMT-based abstraction effectively enumerates all transitions in the abstract state space. As new predicates are added during CEGAR, the

abstract transitions increase exponentially in the worst case. To avoid the exponential complexity of monolithically eliminating the quantifier in (6.1), it is desirable to divide the problem into smaller parts. Unfortunately, there are several factors that impede the subdivision.

- An existential quantifier distributes over disjunctions but not conjunctions. The form of (6.1) is a conjunction, and typically, the concrete transition formula T is also an n-ary conjunction.

- An existential quantifier does distribute over conjuncts that do not share quantified variables (cf. (6.5)–(6.6)). While useful, this simplification has limited applicability to (6.1). To illustrate this, let X be the set of conjuncts of the transition formula that mention a state variable x local to a component, and let Y be the conjuncts that mention a variable y local to another component. One might reason that X and Y are likely to be disjoint. However, if there is any conjunct outside $X \cup Y$ that mentions any quantified variable z occurring in both X and Y, it means that the variables x, y, and z are coupled together and distributing the quantifier over these conjuncts is not legal for precise abstraction. The same applies if there is a longer chain of conjuncts and variables that connects x to y. Moreover, as new predicates are introduced in the CEGAR loop, not only does the abstract state space grow, but typically also more and more state variables become coupled together because they occur in the same predicate.

- In some cases, it is possible to eliminate the quantified variables shared between conjuncts and then push the quantifier into the subformulas. If the variables are Boolean, straightforward application of Shannon's expansion yields

$$\begin{aligned} &\exists xyz : f(a,x,z) \wedge g(a,y,z) \;\equiv \\ &\Big(\big(\exists x : f(a,x,\mathsf{true})\big) \wedge \big(\exists y : g(a,y,\mathsf{true})\big)\Big) \vee \\ &\Big(\big(\exists x : f(a,x,\mathsf{false})\big) \wedge \big(\exists y : g(a,y,\mathsf{false})\big)\Big). \end{aligned}$$

  In the case of real variables that occur in linear (in)equalities, we could apply the Fourier-Motzkin elimination [52] to dispose of the shared variables. However, these eliminations may grow the formula exponentially or worse, and are not investigated in this work.

- An attractive idea would be to start with the cheap Cartesian abstraction and then iteratively drop spurious abstract transitions until the precise abstraction is obtained. Unfortunately, this direction seems infeasible. Due to the asymmetry in the definition of existential abstraction, simple SMT calls can check whether an abstract transition formula is a strict under-approximation or precise, but there seems to be no efficient way to check whether an abstract transition formula allows spurious transitions.

6.2 Exploiting Structure in Abstraction

The general idea in [**VI**] is to apply structural information to divide the monolithic abstraction problem into several smaller problems. Model-level simplifications (Sect. 6.2.1 below) make use of the structure of the system description to construct the predicate abstraction formula in a way that facilitates abstraction. Then, the formula-level simplifications of Sect. 6.2.2 are applied to reduce the complexity of individual quantifier elimination problems. Finally, an efficient quantifier elimination procedure such as the SMT-based approach of Sect. 6.1.2 is applied to the simplified problems in sequence, and these results are joint with Boolean connectives to form the final abstract transition formula.

6.2.1 Model-Level Simplifications

The high-level structural abstraction techniques in [**VI**] divide the abstraction problem based on the finite set of actions that the system can perform, which enables simplifications relying on the fact that individual actions are often local to a part of the system.

Action-wise disjunctive partitioning. In the setting of concurrent systems whose behavior is defined as interleavings of action occurrences, it is natural to compute the abstractions of the actions one at a time. In terms of the precise abstraction formula (6.1), we replace the transition formula T by the restriction of T to one of the actions, going through the actions in sequence. The final abstract transition relation is then the disjunction of

the action-specific abstractions. In essence, the global transition formula is represented as a disjunction with one disjunct per action, and the existential quantifier is distributed over the disjunction. At the same time, the abstraction constraint C_Γ is distributed to every disjunct, but according to the experiments in [**VI**], the potential overhead of this duplication is far outweighed by the benefits of the partitioning.

Exploiting locality. The partitioning allows us to exploit the locality of actions. In particular, if an action is known not to change the value of a state variable x, then we can substitute x for the next-state variable x' when abstracting that action, immediately disposing of one variable x' under the quantifier. Moreover, if a predicate γ_j is such that it does not mention any variable that the action can change, then we know that the value of the corresponding abstract state variable p_j is also not changed by the action. Thus, we can eliminate both p_j and p'_j and their defining predicates from the scope of the quantifier and instead add the frame constraint $p'_j \leftrightarrow p_j$ to the action-specific abstract transition formula.

6.2.2 Formula-Level Simplifications

The four low-level abstraction techniques presented in [**VI**] and described below are designed to further subdivide and simplify the partitioned abstraction problem.

Inlining. After the top-level disjunctive partitioning, the resulting subproblems generally have the form of an existentially quantified n-ary conjunction. By identifying conjuncts of the form $var = expr$, we apply substitution rules that eliminate variables under the quantifier. The exact rewrite rules are listed in [**VI**], and the strategy is to always apply them to the formula until saturation.

In particular, the rewrite rules are designed to work as a part of implementing the locality simplifications described above, assuming that on the model level, the transition formula is constructed with a suitable structure. Defined as a set of generic formula-level transformations, the application of inlining is however not restricted to locality simplifications alone.

Syntactic conjunct clustering. From an n-ary conjunction under an existential quantifier, we identify minimal clusters of conjuncts such that conjuncts that share a quantified variable are in the same cluster. By pushing the quantifier in, each cluster then constitutes a smaller quantifier elimination problem. As discussed in Sect. 6.1.3 above, it is common that all conjuncts collapse into a single cluster, and no benefit is gained. However, the disjunctive partitioning and inlining increase the likelihood that this simplification is effective.

Variable sampling. The syntactic conjunct clustering cannot be applied to the quantifier elimination problem $Q(a) \equiv \exists xyz : f(a,x,z) \wedge g(a,y,z)$ if the shared vector z is non-empty. However, if we fix z to a constant value Z, we can cluster the conjuncts of the resulting problem

$$Q_Z(a) \equiv \exists xy : f(a,x,Z) \wedge g(a,y,Z) \equiv \big(\exists x : f(a,x,Z)\big) \wedge \big(\exists y : g(a,y,Z)\big).$$

For any value Z, we get an under-approximation Q_Z of Q. If Q' is an arbitrary strict under-approximation of Q, any satisfying interpretation of $\neg Q'(a) \wedge f(a,x,z) \wedge g(a,y,z)$ yields a value Z of z that guarantees Q_Z to have at least one satisfying interpretation that does not satisfy Q'. Thus, $Q' \vee Q_Z$ is both an under-approximation of Q and a strict over-approximation of Q'. As the vector of free variables a is Boolean, finitely many values of z are sufficient to completely cover Q as a disjunction of under-approximations.

By variable sampling, we mean the process that starts with $Q' \equiv \mathsf{false}$ and iteratively uses an incremental SMT solver call to find a value Z as above, computes the quantifier-free representation of Q_Z with the help of syntactic conjunct clustering, and replaces Q' by $Q' \vee Q_Z$. This is repeated until Q' and Q are equivalent, that is, until $\neg Q'(a) \wedge f(a,x,z) \wedge g(a,y,z)$ is unsatisfiable.

In the general case, there are several conjuncts under the existential quantifier, and possibly several candidates for a sampled variable or vector of variables z that divides the conjuncts into two or more clusters. Ways to find good candidates for z is left outside the scope of this work. In the experiments in [**VI**], variable sampling is applied with one fixed variable as the sampled variable, namely δ, which denotes the time interval in the time elapse action of hybrid systems.

Blocking don't cares. Consider the SMT-based enumeration in eliminating the quantifier from a formula $\exists x : f(a, x)$ occurring in a context of the form

$$U(a) \vee \big(V(a) \wedge \exists x : f(a, x)\big). \tag{6.8}$$

Here, U and V represent quantifier-free formulas obtained as results from earlier subproblems from the disjunctive and conjunctive partitioning, respectively. We know that any concrete value A of a that satisfies $U(a) \vee \neg V(a)$ is a don't care: no matter whether $\exists x : f(A, x)$ is true or false, the formula (6.8) evaluates to $U(A) \vee V(A)$. We can block such values from the enumeration of satisfying interpretations by restricting the problem $\exists x : f(a, x)$ to $\exists x : f(a, x) \wedge \neg U(a) \wedge V(a)$.

In the experiments of [**VI**], the idea of blocking don't cares is implemented only partially in combination with syntactic conjunct clustering. Namely, when eliminating the quantifiers from a conjunction

$$\big(\exists x : f(a, x)\big) \wedge \big(\exists y : g(a, y)\big)$$

using the SMT-based enumeration, the constraint $f(a, x) \wedge g(a, y)$ is placed to restrict the enumeration of both the left and the right subproblem.

6.3 Results

Table 6.1 shows the effect of the structure-aware approach on the computation time of an abstract transition relation of linear hybrid automata networks. The hybrid models are taken from the HyTech [49] tool distribution, and the abstraction predicates are extracted from the last iteration of a CEGAR loop computation that was separately run using a modified version of NuSMV [21]. Only the 11 hardest instances are listed.

Our baseline is the column labeled "monol.". It denotes the computation time with the SMT-based enumeration using the MathSAT solver [14] on the precise abstraction formula (6.1) with the monolithic transition formula. For the column "partit.", disjunctive partitioning and inlining are applied, and then the subproblems are solved with SMT-based enumeration. In the "clust." column, syntactic conjunct clustering is added to the process.

Finally, the "sampl." column shows the results when all described techniques are in use. In particular, variable sampling is only used in the

Table 6.1. Results of structure-aware abstraction on HyTech models.

	computation time / s			
model	monol.	partit.	clust.	sampl.
active	54.626	18.847	2.410	0.937
active-trace	51.781	22.171	2.473	0.952
audio	13.826	4.547	0.448	0.442
audio-timing	10.910	3.915	0.947	0.690
billiard-timed	0.910	0.732	0.732	1.044
dist-controller	0.320	0.232	0.195	0.147
grc-ver	33.068	19.599	10.421	0.455
new-grc	38.649	17.840	7.395	0.383
railroad	0.170	0.140	0.131	0.112
reactor-clock	0.181	0.133	0.069	0.050
reactor-rect	0.132	0.112	0.051	0.045

subproblem of abstracting the time elapse action, yet it tends to visibly improve the total run time. In all 11 cases, variable sampling further partitions the problem into at least 2 and up to 5 clusters of conjuncts.

The results indicate that our techniques quite consistently accelerate the computation of predicate abstractions with respect to monolithic quantifier elimination. Also, the speed-up is not merely due to any individual simplification but a combination of techniques. Publication [**VI**] presents further analysis of these benchmarks and additional positive results on a set of randomly generated hybrid automata networks.

6.4 Related Work

Computing predicate abstractions with SMT decision procedures was proposed in [64], and combined to BDD-based reasoning in [19]. These works compute a precise predicate abstraction by treating the problem as a monolithic quantifier elimination problem. Either of these approaches can be used in our approach to solve the subproblems resulting from partitioning. In [22], the BDD-based approach is generalized by allowing the formula under the quantifier to be represented as an implicit conjunction of BDDs.

In BDD-based model checking, the symbolic computation of forward images (Eq. (2.2)) is also an instance of a quantifier elimination problem, which has been optimized by e.g. structural partitioning (see [29] and the related work in [**VI**]) and also SAT-based techniques [42].

Giunchiglia et al. [43] present a quantifier elimination procedure that also follows the structure of the formula to push in quantifiers. The rewrite rules in [43] are more aggressive than ours in Sect. 6.2.2 and may cause significant blow-up in the formula size. Quantifier elimination procedures have also been proposed for specific theories such as linear arithmetic [72, 65]. Elimination of nested quantifiers has been optimized in the SMT [73] and purely Boolean [81] context. In our abstraction setting, quantifiers are not nested in the scope of other quantifiers.

Lately, predicate abstraction has been very successfully applied in several works on sequential software verification [7, 28, 50, 69]. An extension to concurrent software is presented in [26], with techniques similar to our disjunctive partitioning and inlining, but specialized to programs in the SpecC language.

6.5 Discussion

Publication [**VI**] brings together disjunctive and conjunctive partitioning and formula rewriting techniques to exploit the structure of the underlying system in computing precise predicate abstractions. In particular, the variable sampling technique employs satisfiability solving technology to gain much of the benefits of Cartesian abstraction without losing precision, and to the author's knowledge, such an approach has not been suggested before.

According to the experiments, the structural partitioning does pay off and reduces computation time. It is not obvious that this should be the case. As pointed out in [**VI**], the formalism of hybrid automata networks encompasses some features that do not agree well with our simplifications, which largely rely on the local nature of actions. Transitions in different hybrid automata can be explicitly synchronized by labels. As a result, one action may involve an arbitrary combination of transitions with the same label. Such synchronizing actions involve control locations and possibly other variables from at least two automata, breaking locality. Also, hybrid automata exhibit location invariants, which are in some cases used as a veiled synchronization mechanism between automata. Finally, the single action that models time elapse is global in nature, which makes it often the most expensive part of the abstraction.

One might consider the combination of structure-aware predicate ab-

straction and step semantics (Sect. 5.1). The parallel ∃-step transition formula purposely mixes the structure and thus defeats the purpose of structural abstraction. However, if one chooses to employ BMC on the abstract model, the idea of the serial ∃-step semantics would be straightforward to apply to the partitioned abstraction: after abstracting each action separately, instead of joining them disjunctively to form the abstract transition formula, put the abstract actions in sequence with intermediate abstract states between them.

With bounded event tracing (Sect. 5.3), a right kind of abstraction is unlikely to be based on state predicates, due to the lack of global states in the encoding. Regarding predicate abstraction for UML models (or object-oriented models in general) and for systems with queues, the efficient computation of abstractions seems secondary, as the bigger question of how to infer semantically meaningful abstraction predicates is still open.

7. Conclusions

This thesis addresses the problem of the computational cost of verifying highly concurrent software-intensive systems. The focus is on checking and falsifying invariant properties of formal design models using symbolic model checking techniques. Contributions are made on four research topics: symbolic model checking of UML state machine models, bounded model checking of systems with queues, symbolic partial order methods for accelerated model checking, and efficient computation of predicate abstraction. In line with our strong emphasis on concurrency, symbolic partial order methods has more weight than the other topics. The proposed techniques enable the use of symbolic model checking to analyze systems modeled with commonly used language features (the topics of UML state machines and systems with queues) and accelerate established model checking techniques (the topics of symbolic partial order methods and structure-aware abstraction).

The main results are

- a formal execution semantics and a compact symbolic transition formula for a subset of the UML 2.0 state machine language,

- several encodings of queues for SMT-based bounded model checking of systems with communication buffers, including UML systems,

- three alternative partial order execution semantics that are shown to significantly accelerate bounded model checking with respect to the conventional interleaving semantics,

- bounded event tracing, a variant of BMC with inherently concurrent semantics, and

- a collection of structure-aware partitioning techniques that are demonstrated to speed up the computation of precise predicate abstractions.

Of these contributions, bounded event tracing requires perhaps the most work to take into use, because the construction of unwindings is still at a preliminary stage.

Lately, testing and pure explicit-state enumeration have been accelerated by tremendous engineering feats, but they cannot keep up with the exponential nature of the state explosion problem. In this work, we propose a purely symbolic approach, where not only the data values but also the control flow is encoded in formulas. This avoids facing up-front any combinatorial explosion due to composing the concurrent components. Instead, the problem is—in the case of bounded model checking—pushed entirely to the satisfiability solver. Apparently, BMC can efficiently find shallow counterexample executions, but it is not strong on properties that require going deep into the reachability graph. Approaches such as abstraction refinement and interpolation improve on blind BMC unrolling by gradually collecting problem-specific information to guide the search. The combination of such approaches to the results of the thesis requires further investigation. A particularly interesting direction is to employ intermediate reachability information in guiding the expansion of unwindings, exploiting the flexibility of the bounded event tracing framework.

Bibliography

[1] Rajeev Alur, Costas Courcoubetis, Thomas A. Henzinger, and Pei-Hsin Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*, pages 209–229. Springer, 1992.

[2] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

[3] Rajeev Alur, Thomas A. Henzinger, and Pei-Hsin Ho. Automatic symbolic verification of embedded systems. *IEEE Transactions on Software Engineering*, 22(3):181–201, 1996.

[4] Nina Amla, Robert P. Kurshan, Kenneth L. McMillan, and Ricardo Medel. Experimental analysis of different techniques for bounded model checking. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003)*, volume 2619 of *Lecture Notes in Computer Science*, pages 34–48. Springer, 2003.

[5] Alessandro Armando, Maria Paola Bonacina, Silvio Ranise, and Stephan Schulz. New results on rewrite-based satisfiability procedures. *ACM Trans. Comput. Log.*, 10(1), 2009.

[6] Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. *Software Tools for Technology Transfer*, 11(1):69–83, 2009.

[7] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In *Integrated Formal Methods (IFM 2004)*, volume 2999 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2004.

[8] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Boolean and Cartesian abstraction for model checking C programs. *Software Tools for Technology Transfer*, 5(1):49–58, 2003.

[9] Jiří Barnat, Luboš Brim, and Petr Ročkai. DiVinE 2.0: High-performance model checking. In *Workshop on High Performance Computational Systems Biology (HiBi 2009)*, pages 31–32. IEEE, 2009.

[10] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability modulo theories. In *Handbook of Satisfiability*, pages 825–885. IOS Press, 2009.

[11] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 1999)*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999.

[12] Nikolaj S. Bjørner. *Integrating Decision procedures for Temporal Verification*. PhD thesis, Stanford University, 1998.

[13] Manfred Broy, Michelle L. Crane, Jürgen Dingel, Alan Hartman, Bernhard Rumpe, and Bran Selic. 2nd UML 2 semantics symposium: Formal semantics for UML. In *MoDELS Workshops 2006*, volume 4364 of *Lecture Notes in Computer Science*, pages 318–323. Springer, 2007.

[14] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani. The MathSAT 4 SMT solver. In *Computer Aided Verification (CAV 2008)*, volume 5123 of *Lecture Notes in Computer Science*, pages 299–303. Springer, 2008.

[15] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. on Computers*, 35(8):677–691, 1986.

[16] Lei Bu, Alessandro Cimatti, Xuandong Li, Sergio Mover, and Stefano Tonetta. Model checking of hybrid systems using shallow synchronization. In *Formal Techniques for Distributed Systems (FMOODS/FORTE 2010)*, volume 6117 of *Lecture Notes in Computer Science*, pages 155–169. Springer, 2010.

[17] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Compututation*, 98(2):142–170, 1992.

[18] Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. CheckFence: checking consistency of concurrent data types on relaxed memory models. In *Programming Language Design and Implementation (PLDI 2007)*, pages 12–21. ACM, 2007.

[19] Roberto Cavada, Alessandro Cimatti, Anders Franzén, Krishnamani Kalyanasundaram, Marco Roveri, and R. K. Shyamasundar. Computing predicate abstractions by integrating BDDs and SMT solvers. In *Formal Methods in Computer-Aided Design (FMCAD 2007)*, pages 69–76. IEEE, 2007.

[20] Søren Christensen and Niels Damgaard Hansen. Coloured Petri Nets extended with place capacities, test arcs and inhibitor arcs. In *Application and Theory of Petri Nets (ATPN 1993)*, volume 691 of *Lecture Notes in Computer Science*, pages 186–205. Springer, 1993.

[21] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV version 2: An opensource tool for symbolic model checking. In *Computer Aided Verification (CAV 2002)*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364. Springer, 2002.

[22] Alessandro Cimatti, Anders Franzén, Alberto Griggio, Krishnamani Kalyanasundaram, and Marco Roveri. Tighter integration of BDDs and SMT for predicate abstraction. In *Design, Automation and Test in Europe (DATE 2010)*, pages 1707–1712. IEEE, 2010.

[23] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981.

[24] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the Association for Computing Machinery*, 50(5):752–794, 2003.

[25] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994.

[26] Edmund M. Clarke, Himanshu Jain, and Daniel Kroening. Verification of SpecC using predicate abstraction. *Formal Methods in System Design*, 30(1):5–28, 2007.

[27] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[28] Edmund M. Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. Predicate abstraction of ANSI-C programs using SAT. *Formal Methods in System Design*, 25(2-3):105–127, 2004.

[29] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.

[30] Fady Copty, Limor Fix, Ranan Fraer, Enrico Giunchiglia, Gila Kamhi, Armando Tacchella, and Moshe Y. Vardi. Benefits of bounded model checking at an industrial setting. In *Computer Aided Verification (CAV 2001)*, volume 2102 of *Lecture Notes in Computer Science*, pages 436–453. Springer, 2001.

[31] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages (POPL 1977)*, pages 238–252. ACM, 1977.

[32] Michelle L. Crane and Jürgen Dingel. Towards a formal account of a foundational subset for executable UML models. In *Model Driven Engineering Languages and Systems (MoDELS 2008)*, volume 5301 of *Lecture Notes in Computer Science*, pages 675–689. Springer, 2008.

[33] Michelle L. Crane and Jürgen Dingel. Towards a UML virtual machine: implementing an interpreter for UML 2 actions and activities. In *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON 2008)*, pages 8:96–8:110. ACM, 2008.

[34] Jori Dubrovin. Checking bounded reachability in asynchronous systems by symbolic event tracing. Technical Report TKK-ICS-R14, Helsinki University of Technology, Department of Information and Computer Science, 2009. `http://ics.tkk.fi/en/research/publications/`.

[35] Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Compututer Science*, 89(4):543–560, 2003.

[36] Javier Esparza and Keijo Heljanko. *Unfoldings — A Partial-Order Approach to Model Checking*. Springer, 2008.

[37] Harald Fecher and Jens Schönborn. UML 2.0 state machines: Complete formal semantics via core state machines. In *Formal Methods: Applications and Technology (FMICS/PDMC 2006)*, volume 4346 of *Lecture Notes in Computer Science*, pages 244–260. Springer, 2007.

[38] Harald Fecher, Jens Schönborn, Marcel Kyas, and Willem P. de Roever. 29 new unclarities in the semantics of UML 2.0 state machines. In *International Conference on Formal Engineering Methods (ICFEM 2005)*, volume 3785 of *Lecture Notes in Computer Science*, pages 52–65. Springer, 2005.

[39] Malay K. Ganai and Aarti Gupta. Accelerating high-level bounded model checking. In *International Conference on Computer-Aided Design (ICCAD 2006)*, pages 794–801. ACM, 2006.

[40] Malay K. Ganai and Aarti Gupta. Completeness in SMT-based BMC for software programs. In *Design, Automation and Test in Europe (DATE 2008)*, pages 831–836. IEEE, 2008.

[41] Malay K. Ganai and Aarti Gupta. Efficient modeling of concurrent systems in BMC. In *International SPIN Workshop (SPIN 2008)*, volume 5156 of *Lecture Notes in Computer Science*, pages 114–133. Springer, 2008.

[42] Malay K. Ganai, Aarti Gupta, and Pranav Ashar. Efficient SAT-based Unbounded Symbolic Model Checking Using Circuit Cofactoring. In *International Conference on Computer-Aided Design (ICCAD 2004)*, pages 510–517. IEEE / ACM, 2004.

[43] Fausto Giunchiglia and Enrico Giunchiglia. Building complex derived inference rules: A decider for the class of prenex universal-existential formulas. In *European Conference on Artificial Intelligence (ECAI 1988)*, pages 607–609. Pitmann Publishing, 1988.

[44] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer, 1996.

[45] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *Computer Aided Verification (CAV 1997)*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 1997.

[46] Orna Grumberg, Flavio Lerda, Ofer Strichman, and Michael Theobald. Proof-guided underapproximation-widening for multi-process systems. In *Principles of Programming Languages (POPL 2005)*, pages 122–131. ACM, 2005.

[47] Helle Hvid Hansen, Jeroem Ketema, Bas Luttik, MohammadReza Mousavi, and Jaco van de Pol. Towards model checking executable UML specifications in mCRL2. *Innovations in Systems and Software Engineering*, 6(1–2):83–90, 2010.

[48] Keijo Heljanko. Bounded reachability checking with process semantics. In *Concurrency Theory (CONCUR 2001)*, volume 2154 of *Lecture Notes in Computer Science*, pages 218–232. Springer, 2001.

[49] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HyTech: a model checker for hybrid systems. *Software Tools for Technology Transfer*, 1:110–122, 1997.

[50] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *Principles of Programming Languages (POPL 2002)*, pages 58–70. ACM, 2002.

[51] Gerard J. Holzmann. *The SPIN Model Checker — primer and reference manual*. Addison-Wesley, 2004.

[52] Jean-Louis Imbert. Fourier's elimination: Which to choose? In *Principles and Practice of Constraint Programming (PPCP 1993)*, pages 117–129, 1993.

[53] Matthias Jantzen and Georg Zetzsche. Labeled step sequences in Petri nets. In *Applications and Theory of Petri Nets and Other Models of Concurrency (Petri Nets 2008)*, volume 5062 of *Lecture Notes in Computer Science*, pages 270–287. Springer, 2008.

[54] Kurt Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods, and Practical Use*, volume 1. Springer, 1997.

[55] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Comput. Surv.*, 41(4), 2009.

[56] Toni Jussila. On bounded model checking of asynchronous systems. Research Report A97, Helsinki University of Technology, Laboratory for Theoretical Computer Science, 2005. Doctoral dissertation.

[57] Toni Jussila, Jori Dubrovin, Tommi Junttila, Timo Latvala, and Ivan Porres. Model checking dynamic and hierarchical UML state machines. In *Model Development, Validation and Verification; 3rd International Workshop (MoDeV^{2}a 2006)*, pages 94–110, 2006.

[58] Toni Jussila, Keijo Heljanko, and Ilkka Niemelä. BMC via on-the-fly determinization. *Software Tools for Technology Transfer*, 7(2):89–101, 2005.

[59] Vineet Kahlon, Chao Wang, and Aarti Gupta. Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In *Computer Aided Verification (CAV 2009)*, volume 5643 of *Lecture Notes in Computer Science*, pages 398–413. Springer, 2009.

[60] Roope Kaivola. Formal verification of Pentium 4 components with symbolic simulation and inductive invariants. In *Computer Aided Verification (CAV 2005)*, volume 3576 of *LNCS*, pages 170–184. Springer, 2005.

[61] Victor Khomenko, Alex Kondratyev, Maciej Koutny, and Walter Vogler. Merged processes: a new condensed representation of Petri net behaviour. *Acta Inf.*, 43(5):307–330, 2006.

[62] Daniel Kroening and Ofer Strichman. Efficient computation of recurrence diameters. In *Verification, Model Checking, and Abstract Interpretation (VMCAI 2003)*, volume 2575 of *Lecture Notes in Computer Science*, pages 298–309. Springer, 2003.

[63] Robert P. Kurshan. Verification technology transfer. In *25 Years of Model Checking*, volume 5000 of *Lecture Notes in Computer Science*, pages 46–64. Springer, 2008.

[64] Shuvendu K. Lahiri, Robert Nieuwenhuis, and Albert Oliveras. SMT techniques for fast predicate abstraction. In *Computer Aided Verification (CAV 2006)*, volume 4144 of *Lecture Notes in Computer Science*, pages 424–437. Springer, 2006.

[65] Rüdiger Loos and Volker Weispfenning. Applying linear quantifier elimination. *Comput. J.*, 36(5):450–462, 1993.

[66] Antoni W. Mazurkiewicz. Trace theory. In *Advances in Petri Nets*, volume 255 of *Lecture Notes in Computer Science*, pages 279–324. Springer, 1986.

[67] Kenneth L. McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In *Computer Aided Verification (CAV 1992)*, volume 663 of *Lecture Notes in Computer Science*, pages 164–177. Springer, 1993.

[68] Kenneth L. McMillan. Interpolation and SAT-based model checking. In *Computer Aided Verification (CAV 2003)*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2003.

[69] Kenneth L. McMillan. Lazy abstraction with interpolants. In *Computer Aided Verification (CAV 2006)*, volume 4144 of *Lecture Notes in Computer Science*, pages 123–136. Springer, 2006.

[70] Kenneth L. McMillan. Lazy annotation for program testing and verification. In *Computer Aided Verification (CAV 2010)*, volume 6174 of *Lecture Notes in Computer Science*, pages 104–118. Springer, 2010.

[71] Kenneth L. McMillan and Nina Amla. Automatic abstraction without counterexamples. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003)*, volume 2619 of *Lecture Notes in Computer Science*, pages 2–17. Springer, 2003.

[72] David Monniaux. A quantifier elimination algorithm for linear real arithmetic. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2008)*, volume 3835 of *Lecture Notes in Computer Science*, pages 243–257. Springer, 2008.

[73] David Monniaux. Quantifier elimination by lazy model enumeration. In *Computer Aided Verification (CAV 2010)*, volume 6174 of *Lecture Notes in Computer Science*, pages 585–599. Springer, 2010.

[74] Robert Nieuwenhuis and Albert Oliveras. Fast congruence closure and extensions. *Information and Computation*, 205:557–580, 2007.

[75] Object Management Group. *Semantics of a Foundational Subset for Executable UML Models, Version 1.0 Beta 3*, 2010. `http://www.omg.org/spec/FUML/`.

[76] Object Management Group. *UML 2.3 Superstructure*, 2010. `http://www.omg.org/spec/UML/2.3/`.

[77] Shougo Ogata, Tatsuhiro Tsuchiya, and Tohru Kikuno. SAT-based verification of safe Petri nets. In *Automated Technology for Verification and Analysis (ATVA 2004)*, volume 3299 of *Lecture Notes in Computer Science*, pages 79–92. Springer, 2004.

[78] Sam Owre, John Rushby, N. Shankar, and David Stringer-Calvert. PVS: an experience report. In *International Workshop on Current Trends in Applied Formal Methods (FM-Trends 1998)*, volume 1641 of *Lecture Notes in Computer Science*, pages 338–345. Springer, 1998.

[79] Corina S. Pasareanu and Willem Visser. Symbolic execution and model checking for testing. In *Haifa Verification Conference (HVC 2007)*, volume 4899 of *Lecture Notes in Computer Science*, pages 17–18. Springer, 2007.

[80] Radek Pelánek. BEEM: Benchmarks for explicit model checkers. In *International SPIN Workshop (SPIN 2007)*, volume 4595 of *Lecture Notes in Computer Science*, pages 263–267. Springer, 2007.

[81] David A. Plaisted, Armin Biere, and Yunshan Zhu. A satisfiability procedure for quantified Boolean formulae. *Discrete Applied Mathematics*, 130(2):291–328, 2003.

[82] Mukul R. Prasad, Armin Biere, and Aarti Gupta. A survey of recent advances in SAT-based formal verification. *Software Tools for Technology Transfer*, 7(2):156–173, 2005.

[83] Ishai Rabinovitz and Orna Grumberg. Bounded model checking of concurrent programs. In *Computer Aided Verification (CAV 2005)*, volume 3576 of *Lecture Notes in Computer Science*, pages 82–97. Springer, 2005.

[84] Lukman Ab Rahim and Jon Whittle. Verifying semantic conformance of state machine-to-Java code generators. In *Model Driven Engineering Languages and Systems (MoDELS 2010)*, volume 6394 of *Lecture Notes in Computer Science*, pages 166–180. Springer, 2010.

[85] Silvio Ranise and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.smt-lib.org`, 2011.

[86] Jussi Rintanen, Keijo Heljanko, and Ilkka Niemelä. Planning as satisfiability: Parallel plans and algorithms for plan search. *Artificial Intelligence*, 170(12–13):1031–1080, 2006.

[87] Timm Schäfer, Alexander Knapp, and Stephan Merz. Model checking UML state machines and collaborations. *Electr. Notes Theor. Comput. Sci.*, 55(3), 2001.

[88] Jens Schönborn and Marcel Kyas. Refinement patterns for hierarchical UML state machines. In *Fundamentals of Software Engineering (FSEN 2009)*, volume 5961 of *Lecture Notes in Computer Science*, pages 371–386. Springer, 2009.

[89] Viktor Schuppan and Armin Biere. Efficient reduction of finite state model checking to reachability analysis. *Software Tools for Technology Transfer*, 5(2-3):185–204, 2004.

[90] Stefan Schwoon. *Model-Checking Pushdown Systems*. PhD thesis, Technische Universität München, 2002.

[91] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In *Formal Methods in Computer-Aided Design (FMCAD 2000)*, volume 1954 of *Lecture Notes in Computer Science*, pages 108–125. Springer, 2000.

[92] The SMUML software distribution version 1.0.1, 2008. Software. `http://www.tcs.hut.fi/Research/Logic/SMUML.shtml`.

[93] SRI International. Yices 2.0 prototype, 2009. Software.

[94] Aaron Stump, Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. A decision procedure for an extensional theory of arrays. In *Logic in Computer Science (LICS 2001)*, pages 29–37. IEEE, 2001.

[95] Dejvuth Suwimonteerabuth, Javier Esparza, and Stefan Schwoon. Symbolic context-bounded analysis of multithreaded Java programs. In *International SPIN Workshop (SPIN 2008)*, volume 5156 of *Lecture Notes in Computer Science*, pages 270–287. Springer, 2008.

[96] Chao Wang, Zijiang Yang, Vineet Kahlon, and Aarti Gupta. Peephole partial order reduction. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*, volume 4963 of *Lecture Notes in Computer Science*, pages 382–396. Springer, 2008.

[97] Martin Wehrle and Jussi Rintanen. Planning as satisfiability with relaxed $\exists$-step plans. In *Advances in Artificial Intelligence (AI 2007)*, volume 4830 of *Lecture Notes in Computer Science*, pages 244–253. Springer, 2007.

[98] Benjamin Weiß. Predicate abstraction in a program logic calculus. *Sci. Comput. Program.*, 76(10):861–876, 2011.

DISSERTATIONS IN INFORMATION AND COMPUTER SCIENCE

TKK-ICS-D14 Hirsimäki, Teemu.
Advances in Unlimited-Vocabulary Speech Recognition for Morphologically Rich Languages. 2009.

TKK-ICS-D15 Heikinheimo, Hannes.
Extending Data Mining Techniques for Frequent Pattern Discovery: Trees, Low-Entropy Sets, and Crossmining. 2010.

TKK-ICS-D16 Hermelin, Miia.
Multidimensional Linear Cryptanalysis. 2010.

TKK-ICS-D17 Savia, Eerika.
Mutual Dependency-Based Modeling of Relevance in Co-Occurrence Data. 2010.

TKK-ICS-D18 Liitiäinen, Elia.
Advances in the Theory of Nearest Neighbor Distributions. 2010.

TKK-ICS-D19 Lahti, Leo.
Probabilistic Analysis of the Human Transcriptome with Side Information. 2010.

TKK-ICS-D20 Miche, Yoan.
Developing Fast Machine Learning Techniques with Applications to Steganalysis Problems. 2010.

TKK-ICS-D21 Sorjamaa, Antti.
Methodologies for Time Series Prediction and Missing Value Imputation. 2010.

TKK-ICS-D22 Schumacher, André
Distributed Optimization Algorithms for Multihop Wireless Networks. 2010.

Aalto-DD99/2011 Ojala, Markus
Randomization Algorithms for Assessing the Significance of Data Mining Results. 2011

Aalto-DD 111/2011

As increasingly critical tasks are being handed over to software-intensive systems, their reliability is becoming more and more essential. At the same time, the verification of computer systems is becoming more demanding due to their distributed nature and the complexity of individual components. The doctoral dissertation of Jori Dubrovin adapts symbolic model checking techniques to the challenging problem of verifying critical safety properties of concurrent systems with software features, expressed in an industrial modeling language such as UML. The work has foundations in verification and concurrency theory, with attention to practical issues, in particular, ways to perform the computationally heavy verification task efficiently using modern logic solvers.

ISBN 978-952-60-4349-4 (pdf)
ISBN 978-952-60-4348-7
ISSN-L 1799-4934
ISSN 1799-4942 (pdf)
ISSN 1799-4934

Aalto University
School of Science
Department of Information and Computer Science
www.aalto.fi